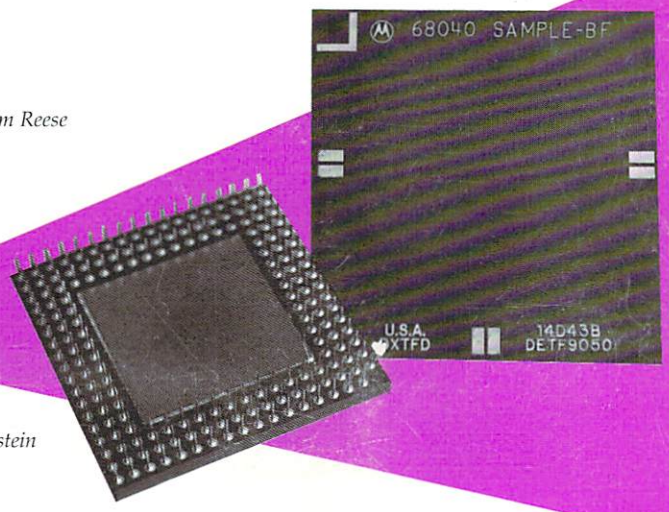
Disk Inside

source code and executables for all articles

AMIGA WORLD

TECH JOURNAL

ARTICLES



Look inside Motorola's MC68040 chip; see page 2.

REVIEWS

COLUMNS

ON DISK

(See page 33.)

Loads of Libraries

Custom Printer Drivers

Plus source code and executables for articles

0 74470 12070 4 12

U.S.A. $15.95
Canada $19.95

Volume 1, Number 5 | December 1991 | An IDG Communications Publication

SAVE IT. MOVE IT. GET IT BACK.

Back-Up...Transfer...Retrieve Quickly And Easily With Central Coast's Software For The Amiga

Valuable utility programs can save you time, money and, in the case of catastrophic errors like hard drive failure, possibly months of work.

Quarterback Tools – Recover Lost Files

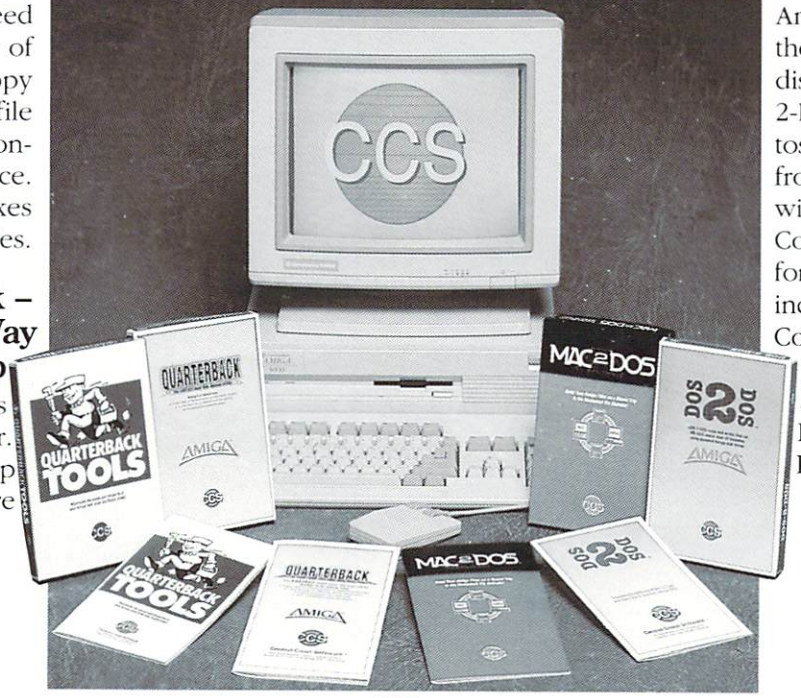
Fast and easy. Reformats all types of disks – either new or old filing systems – new or old Workbench versions. Also optimizes the speed and reliability of both hard and floppy disks. Eliminates file fragmentation. Consolidates disk space. Finds and fixes corrupted directories.

Quarterback – The Fastest Way To Back-Up

Backing-up has never been easier. Or faster. Back-up to, or restore from: floppy disks, streaming tape (AmigaDOS-compatible), Inner-Connection's Bernoulli drive, or ANY AmigaDOS-compatible device.

Mac-2-Dos & Dos-2-Dos – A Moving Experience

It's easy. Transfer MS-DOS and ATARI ST text and data files to-and-from AmigaDOS using the Amiga's own disk drive with Dos-2-Dos; and Macintosh files to-and-from your Amiga with Mac-2-Dos. Conversion options for Mac-2-Dos include ACSII, No Conversion, MacBinary, Postscript, and Mac-Paint to-and-from IFF file format.



Central Coast Software
A Division Of New Horizons Software, Inc.

206 Wild Basin Road, Suite 109, Austin, Texas 78746
(512) 328-6650 • Fax (512) 328-1925

Quarterback Tools, Quarterback, Dos-2-Dos and Mac-2-Dos are all trademarks of New Horizons Software, Inc.

Circle 3 on Reader Service card.

The AmigaWorld Tech Journal (ISSN 1054-4631) is an indepedent journal not connected with Commodore Business Machines, Inc. *The AmigaWorld Tech Journal* is published bimonthly by IDG Communications/Peterborough, Inc., 80 Elm St., Peterborough, NH 03458. U.S. Subscription rate is $69.95, Canada and Mexico $79.95, Foreign Surface $89.95, Foreign Airmail $109.95. All prices are for one year. Prepayment is required in U.S. funds drawn on a U.S. bank. Application to mail at 2nd class postage rates is pending at Peterborough, NH and additional mailing offices. Phone: 603-924-0100. **Postmaster**: Send address changes to *The AmigaWorld Tech Journal*, 80 Elm St., Peterborough, NH 03458. *The AmigaWorld Tech Journal* makes every effort to assure the accuracy of articles, listings and circuits published in the magazine. *The AmigaWorld Tech Journal* assumes no responsibility for damages due to errors or omissions. Third class mail enclosed.

Bulk Rate
US Postage Paid
IDG Communications/Peterborough, Inc.

MESSAGE PORT

The survey says...

WHEN YOU START a new magazine, you guess. Sure, you base your theory on some data from potential readers, but you still gamble and hope for the best. That's in large part what we did when we launched *The AmigaWorld Tech Journal*...we hoped for the best. Were we on the mark? You delivered the answer in the results of our recent subscriber survey. Oh, we're not perfect yet. 17% say we're not technical enough (of course, 15% say we're too technical), but that 60% of you that said we're "just right" made my day! Thanks.

So, who reads *The AmigaWorld Tech Journal*? 11% are beginners, 40% are intermediates, 32% are advanced, and 17% are developers. 51% of you own Amiga 2000s or 2500s, 20% powered up to A3000s, 23% have A500s, and a loyal 29% still use A1000s. Not surprisingly, 42% of *Tech Journal* readers use their Amigas primarily for software development. Multisync monitors, CD-ROM drives, accelerators, and programming tools top your shopping lists.

As for languages, C is the favorite, with SAS outscoring Manx as the compiler of choice. 78% of you want heavy C coverage, and 97% want at least moderate coverage. Assembly follows: 32% called for heavy coverage, 76% for moderate or more. ARexx scores third (75% moderate to heavy), BASIC comes in low: 60% think it deserves only light attention.

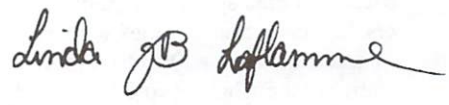
The majority of you (84%) prefer to have the code examples on disk. One respondent went so far as to declare this the "best part of the *Journal*." "Let most of the on-disk code reflect professional-level programming. We amateurs want to know how to do it right," another reader requested. The Peer Review Board is dedicated to ensuring that this is the case.

A 64% majority wants us to go monthly. (Yipes.)

Reviews are a hot topic of debate. "With only 48 pages to fill [now 64], reviews are better kept elsewhere," one reader recommended. Most of you seconded this motion, indicating if we have a choice between printing reviews or one more tutorial article with code, we should run the tutorial. (I agree, motion carried.) When we do print reviews, however, most of you prefer in-depth examinations of programming tools, followed by performance tests of hardware.

As to articles, specific techniques, OS coverage, and algorithms are the most requested types. More hardware coverage was a favorite comment. One respondent proposed a series of articles that describe the Amiga's various processors—basic chip set, enhanced chip set, named chips, 68000, 68020, '030, and 68882. As you read this, the project is already in the works. To the reader who called for an explanation of IFF85, we'll go you one better: We're teaming up with CATS for an article on the latest version of the venerable standard. And no, we won't forget you beginners. In each issue we try to include at least one introductory piece and provide background sidebars on the weightier subjects.

One thing we'll always do is listen to your comments and suggestions. Don't wait for another survey to let us know what you think. ■



68030 to 68040 Differences

What's new to Motorola's latest generation of chip?

By John Meek and Tim Reese

A HIGH PERFORMANCE 32-bit microprocessor in Motorola's 68000 family, the 68040 maintains 100% user-code compatibility with previous members of the family. The '040's Integer Unit, Floating-Point Unit, and Memory Subsystem operate in parallel to achieve significantly higher performance than that of its predecessor. Motorola claims four times the performance of a 68020 microprocessor at the same clock speed. In practice, we have seen performance of about two to four times that of a 68030 (at the same clock rate). What is this 68040 and what makes it so much faster than the '020 and '030? To answer this question, we will examine some of the major philosophical and architectural differences between the '040 and the '030. (Consult the *MC68040 32-Bit Microprocessor User's Manual* and the *MC68040 Designer's Handbook* for complete details regarding issues not discussed here.)

OPTIMIZATION AND SPEED ENHANCEMENTS

What kind of changes are necessary to double or quadruple the speed of a family of processors? Motorola's approach was to collect a rather formidable amount of data based on existing 68020 and 68030 systems. Millions of cycles of bus activity were recorded on various systems running different applications and different operating systems. This trace data was then analyzed using a high-level statistical performance model along with cache and MMU simulators.

Motorola's engineers based their architectural decisions on the '040 performance goals, the collected trace data, customer input, ease of implementation, and, of course, silicon usage. The instructions with the highest dynamic execution frequency were optimized. To maximize the execution speed of these newly optimized instructions, they pipelined the Integer Unit. The most common instructions execute in a single clock cycle. The cycle time of the ALU is an important factor in optimizing instructions. To achieve high performance, it is very desirable for the peak instruction rate to be equal to that of the ALU cycle rate. With this in mind, the '040 is designed so that the ALU cycle rate is matched to the cache access time. The Floating-Point Unit was also optimized and pipelined. Of more interest, however, is the fact that the Floating-Point Unit now resides on-chip. This arrangement greatly reduces the amount of time spent in external arbitration to a coprocessor.

Motorola's analysis also indicated that the cache architecture should be based on the Harvard model (separate address and data space in memory), as in the '030. The designers, however, changed the caching arrangement from directly mapped (as on the '030) to a set-associative type. The associative cache scheme provides a much higher hit rate than the direct mapped type (see "Cache Mapping Techniques"). In addition, the copy-back style of data caching was incorporated into the new design. It has been observed that, depending on the program flow, copy-back caching may increase performance by as much as 25% over conventional write-through cache schemes.

Finally, you should understand that the architecture is intended to provide most of the memory bandwidth from the internal caches, meaning accesses to external memory are optimized for the loading and unloading of the internal cache systems. The bus protocol has been simplified to efficiently handle cache line bursts and to minimize the latency from a cache miss. The simple synchronized start-terminate sequence also simplifies the hardware interface to external devices.

Arguably, the biggest difference between the '030 and the '040 is the bus structure. This includes termination, dynamic bus sizing, transfer attributes versus function codes, burst retry, and the output buffers.

DATA BUS PROTOCOL

The data bus protocol has been completely changed from the previous processors. To optimize the pipelines, the '040 was designed to be a fully synchronous processor, whereas the MC68030 could be used either as a synchronous or an asynchronous processor. In the past, the 68K family has used AS* and DS* to indicate that the information on the bus is valid upon assertion. The '040 deviates from this type of data bus protocol by introducing a signal called Transfer Start: TS*. Transfer Start, alone, does not indicate that the information on the address and data bus is valid, but rather that the information is valid on the next rising edge of BCLK. This will help hardware designers as the clock speeds increase for the '040, and it allows you to maximize the cycles so that extra cycles are not taken up for such things as decoding and control when the clock cycles get faster.

The termination of the cycle is different as well. The MC68030 provides two distinct ways to terminate a cycle. The bus cycle termination signals DSACKx* and STERM* both provide asynchronous or synchronous termination. For terminating a cycle with the '040, Motorola eliminated the DSACKx and STERM signals and introduced a new signal called Transfer Acknowledge (TA*). The '040 uses only the TA* signal to terminate a cycle. Like the TS* signal, the TA* signal is synchronous. It is only valid with the rising edge of the BCLK. For systems that absolutely need to operate the data bus asynchronously, Motorola provides the data latch enable mode. When you select this mode at reset time all data will be read by the '040 processor with the Data Latch

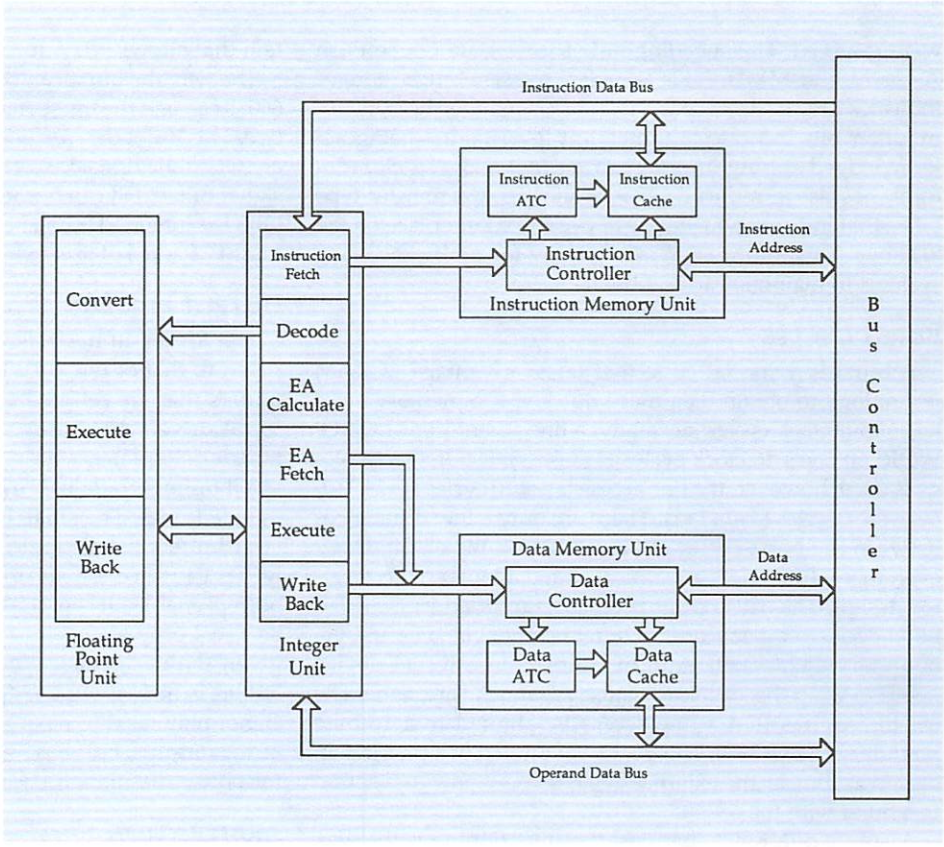


Figure 1: An illustration of the 68040's data transfer paths.

Enable (DLE*) signal, thus providing for an asynchronous read. The cycle, however, is still terminated by TA*.

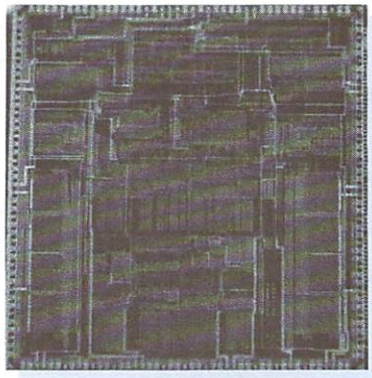
The goal of designing any adapter or accelerator board is to match the host processor functions with the accelerated processor functions. Asynchronous termination can be a problem if you do not take special care. If the processor is running in both a synchronous and an asynchronous environment, then the data latch enable mode may not be appropriate. In this case, the asynchronous termination must be synchronized or it could create metastability problems. At the higher speeds, the problem gets worse. Conversely, if signals are synchronized that don't need to be, then cycles are wasted on the front and back end of the synchronization process. Suffice to say that this takes some thought in an adapter design.

DYNAMIC BUS SIZING

Unlike the MC68020 and MC68030, the MC68040 no longer supports dynamic bus sizing. To understand how this affects '040 design, you must know what dynamic bus sizing is (see "Dynamic Bus-Sizing Details").

The '040 has no dynamic bus-sizing capability. If the processor requires a byte down a particular byte lane, then the byte has to be returned to the processor via the correct byte lane. This forces byte devices to either multiplex their byte data onto the correct byte lane or to access through software the exact byte lane by control of the address. Probably, there is more hardware and control associated with matching an '040 bus structure to a fellow member 68K processor than any other aspect of adapter design.

Another interesting aspect to dynamic bus sizing is an '030 processor will take care of completing four or two accesses, respectively, to a device if the processor makes a LONGWORD or WORD request to a byte port. This makes accessing a device very easy for the software, because all of the work is done by a device such as the '030. It actually can eliminate the need for the software to even know what size the data port is on an external device. For the '040, because the processor does not know about such things as 8- or 16-bit ports, the processor reads as much data as requested upon termination of the cycle. Therefore, it can be disasterous to request a LONGWORD of information for a byte port because only one byte of data will be correct.



A look inside the '040 chip. Note the large Integer Unit in the center and the Floating-Point Unit below it. Bracketing these on the left and right are the Instruction Cache and the Data Cache, respectively. Above the caches are the Instruction and Data ATCs. The Bus Controller is in the top right corner.

In adapter design, the challenge is to make the '040 operate exactly like the 68030 for dynamic bus operation. This means external control initiates and terminates cycles independent of the 68040 so that all of the data is present and is on the correct byte lanes.

TRANSFER ATTRIBUTES VS. FUNCTION CODES

For the 68040, function codes are eliminated. Previously, the 68K family used function codes to indicate the address space type for a bus cycle: supervisor, user or CPU, and program versus code space. Some designs have used these bits and others have completely neglected their use in the system design. The 68040 replaces the function-code bits with the transfer-modifier and transfer-type bits. The three transfer-type bits are essentially the same as the old function-code bits with some additional encoding where the function-code bits ►

were reserved. The encoding now shows Data Cache Push Accesses and MMU Table Search Accesses, but no longer indicates CPU address space. In another change, the transfer-modifier bits indicate the interrupt level that is being acknowledged during an interrupt-acknowledge cycle. Motorola has also introduced two new signals—the transfer-type bits—that indicate which type of cycle is taking place on the '040 bus. This information is necessary for multiprocessing systems to maintain cache coherency.

BURST CYCLES

A burst is a special cycle that takes advantage of RAM technology to obtain data more quickly. If a memory cycle takes four clock cycles, normally, a line access (four LONGWORDS) takes 16 clock cycles (4+4+4+4). If burst technology is used, however, the line access takes 10 cycles (4+2+2+2) or even 7 cycles (4+1+1+1). There are only a few differences between the burst cycles of the 68040 and the 68030. However, these differences are critical in any adapter or accelerator design. The 68030 can handle a wrap of address with a burst cycle. In other words, a burst need not be on a 16-byte boundary (A2/A3≠0). And of course, the external device needs to wrap the address because A2 and A3 are static. The 68040 requires that a line access be aligned to a 16-byte boundary (A2/A3=0). This really isn't much of a problem in design as long as the external devices know how to respond to a burst with the 68040.

A subtle difference that deserves mentioning is the way retry is done on burst between the 68040 and the 68030. The 68030 can retry on any access during a burst. The 68040 can only retry on the first access on a burst. This can cause a great deal of trouble in designing an '040 adapter for a system that allows retry during any cycle in a burst.

The way the burst can be inhibited is altered, as well. The 68030 can end the burst on any access if the CIIN becomes active. This is not the case with the 68040. The 68040 only looks at TCI on the first access and doesn't affect the burst. A new signal called TBI can become active on the first cycle and inhibit the intended burst. Note, after the first cycle in a line access, TCI and TBI are both ignored by the 68040 processor.

OUTPUT BUFFERS

The MC68040 has a flexible output buffer scheme compared to that of the '030. The '040 has two different output buffers that are selected at reset. The large buffers contain 55mA drivers that are designed to drive 50-ohm terminated transmission lines with a minimum amount of delay. The small buffers contain 5mA drivers designed to drive unterminated lines. The small buffers have more delay than the large buffers but dissipate significantly less power. Normally, the '040 can dissipate up to 3 watts, but with all of the large buffers enabled it can dissipate 5 watts of power. This could be a real problem in some designs. Therefore, Motorola has given the designer the flexibility of choosing the output buffers in three groups to optimize power versus speed. A designer may need to have large buffers enabled on a portion of the outputs, but may want the others to be small buffers for power dissipation reasons.

ARBITRATION

Unlike current members of the 68000 family of processors, the 68040 provides no on-chip arbitration for the external bus. The '040 was designed to be a slave with an external arbiter controlling all of the bus arbitration. This scheme pro-

Dynamic Bus-Sizing Details

Two terms must defined for us to discuss dynamic bus sizing. Motorola extensively uses the term *port* when talking about dynamic bus sizing. A port is nothing more than a device with a set data-bus size. For example, an eight-bit peripheral would be considered an eight-bit port, a 16-bit peripheral or any external device that has 16 bits is considered a 16-bit port, and so forth. The second term is *byte lane*. A 32-bit bus, of course, has four byte lanes. Each lane is distinct (31:24, 23:16, 15:8, 7:0), and an address is associated with each byte (see Figure 2). Therefore, the processor, depending on the instruction, expects data on particular byte lanes.

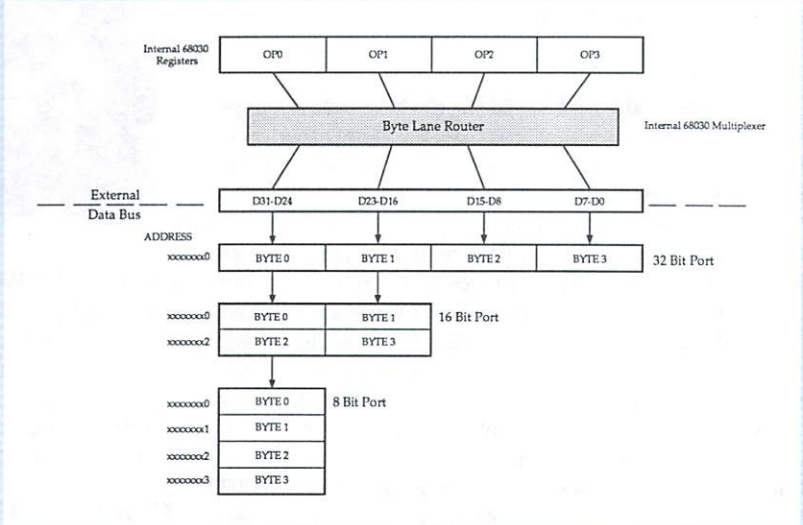


Figure 2: The four byte lanes of a 32-bit bus.

Dynamic bus sizing requires that the byte or word port be fixed on set byte lanes. This actually is helpful for a hardware designer because the processor can request bytes from any of the byte lanes (as per the address being requested by the current instruction), and the designer can give the bytes on the set byte lanes to which the device is attached. Through dynamic bus sizing, the processor then dynamically puts the byte on the correct byte lane for the instruction being processed. This, of course, is advantageous to a device that can only control a single byte lane. □

—JM and TR

vides more flexibility in the design of board-level arbitration. The arbiter may be set up to mimic 68000-style processors (in which the 68040 is the de facto bus master), or it may treat the 68040 as one of many potential bus masters. This has given us a lot of flexibility in our design to do exactly what we want to do in terms of priority and speed with our arbitration system. The '040 now has to request the bus (BR*) any time that any external transfer takes place. The external arbiter then grants the bus to the '040 (BG*), and the '040 responds with (BB*). The largest implication of moving the arbiter from internal to external is that now you can have multiple '040s on the same bus. With the new scheme and because bus arbitration is synchronous, it is possible to switch processors in a single clock cycle. This allows for a convenient strategy in the future for multiprocessor systems.

"The '040 has two separate MMUs and two 4K caches (one for instructions and one for data)."

MEMORY SUBSYSTEM

The 68040 memory subsystem must provide instructions and data to the Integer and Floating-Point Units at a sufficient rate to match their performance. Keeping these units fed from a single external bus is not possible, so the memory unit is designed to provide the majority of the required bandwidth from the internal caches. The external bus is optimized to handle cache loading and unloading. To this end, the system is Harvard architecture (as in the 68030).

Beyond the major architecture, the '030 and '040 subsystems diverge. The large difference is mainly attributed to the performance increase required of the '040. The '040 has two separate memory management units (MMUs) and two 4K caches (one each for instruction and data). The '030 also has two separate (smaller) cache units, but only a single MMU. In the '040, each MMU provides a separate address translation cache (ATC) of 64 entries and a pair of transparent translation registers (TTR) that operate in parallel with the caches. The MMUs supply full (32-bit) logical-to-physical address translation providing for complete memory management in a virtual, demand-paged environment. The caches in the '030 are addressed logically as opposed to the '040's physically addressed caches. The reasoning behind this is explained in the Caches section below. Each page (4K or 8K) may independently specify cache modes, write protection, and supervisor/user protection. The TTRs may specify the same attributes without translating (transparent translation) addresses for memory segments on 16 megabyte boundaries.

The large minimum page size (relative to the '030) is a result of the fact that ATC and physical cache lookups can occur at the same time. This means, of course, that the same bits cannot be used for both lookups. The reduction in the available bits increases the minimum page size. Some simple "four-way set-associative 4K byte cache" math says: Minimum page size=2**10=1K byte (the proof is left to you...). Looking to the future, Motorola elected to increase this size in anticipation of larger caches. The debate over minimum page size, however, rages on. Most existing systems use a 4K page size, while the newer ones use 8K. Larger page sizes reduce ATC faults and increase paging efficiency to disks. Unfortunately, they also increase internal fragmentation (thereby boosting the odds that unused data is unnecessarily paged to and from disk).

Fortunately, the user is relieved of the responsibility of dealing with any of these MMU-related issues (one of the reasons why user code remained 100% compatible with the 68030). Supervisor-only instructions are provided to maintain the TTRs and ATCs. Separate ATCs may be maintained for Supervisor and User spaces. The ATC-loading algorithm (a hardware tablewalk) uses separate User and Supervisor root pointers as entry points into the three-level table structure of descriptors. For additional information regarding this table structure, refer to the *MC68040 32-Bit Microprocessor User's Manual*.

There are a number of other Supervisor-level differences between the memory management philosophies of the two chips. As these issues are not of general interest, they will not be discussed here.

CACHES

The cache system, while retaining the same major architecture as the '030, has some very significant differences and enhancements. A separate bus controller loads and unloads caches by burst transfer. The use of a separate bus controller, along with the MMU and cache controllers, allows the internal caches to provide the majority of the required memory bandwidth (cache lookups and external cycles may occur simultaneously).

Cache size grew from 256 bytes in the MC68030 to 4096 bytes in the MC68040. The MC68040 retained the 68030's Harvard-style architecture, providing separate 4K caches for instructions and data. In the '040, each cache is organized as four-way set-associative with 64 sets of four lines each (for explanation, see "Cache Mapping Techniques"). Each line contains 16 bytes of data or instructions, an address tag field, and state information. This is contrasted with the direct-mapped cache organization of the '030. Motorola engineers chose a four-way set-associative organization to minimize silicon area and maximize hit rates (a fully associative cache would actually provide greater hit rates, but the required silicon real estate was prohibitive). Motorola also studied other organizations (direct-mapped, two-way set-associative), but decided on four-way set-associative for its superior performance in a limited space.

As in the '030, the instruction cache is limited to two modes of operation: cacheable and noncacheable. The data cache, however, has grown to four modes of operation: cacheable copy-back, cacheable write-through, noncacheable nonserialized, and noncacheable serialized. Copy-back caching produces maximum performance and minimum bus utilization in single-processor implementations. For multiple processors, the proper use of copy-back requires complicated bus and cache state protocols to maintain cache coherency. This is one of the reasons that write-through caching was also supported on the MC68040.

Both noncacheable modes operate such that the data cache is never used or loaded. Any data operation directly accesses external memory. Noncacheable serialized mode simply ►

guarantees that all reads and writes will occur in program sequence. This mode may sometimes be necessary due to the pipeline nature of the Integer Unit (IU). The IU may actually reorder accesses to maximize performance.

Operation of the cacheable modes is a bit different than in the '030. For both cacheable modes, a read miss causes a line to be loaded into the cache. Writes are handled differently, depending on the selected mode for that page. A write to a write-through page will always update the external memory and will update the cache if the operand is resident. This is the same type of data caching supported by the MC68030. An important difference between the '030 and '040 write-through cache involves cache-line allocation. The '040 operates under a no-write-allocate policy (writes that miss in the cache are written to external memory, but the corresponding line is not loaded into the cache). Write allocation in the '030 is programmable via a bit in the Cache Control Register.

In copy-back mode, a write miss to a copy-back page will allocate a cache entry for loading, then load it from external memory. The write data is then written to the cache only. Because the operand value in external memory no longer matches the value in the cache, the entry is considered dirty. Dirty entries are written to external memory in the event of cache flush or line replacement (the cache is full, and a new line must be loaded). If the cache is full, a pseudo-random algorithm designates the cache line to be replaced. This line is then moved to an internal temporary buffer and tagged for update (it is placed in the write queue and will be written to the external memory in its turn). In the event of a write hit to a copy-back page, the cache line is updated, and the dirty bits are set for the appropriate LONGWORDS within the line. Copy-back data caching is not available on the MC68030.

Cache-line loading is part of the newly optimized memory subsystem in the '040. When a new cache line is required, the cache controller requests a line read from the bus controller. The bus controller requests a burst-read transfer by indicating a line access on the size signals (the '030 asserts the signal CBREQ). The responding device may indicate that burst transfers are not supported by asserting the Transfer Burst Inhibit (TBI) signal. Similarly, an '030 system may inhibit a burst transfer by failing to supply CBACK (Cache Burst ACKnowledge) with its standard termination. Bursting on the '030 may also be controlled by setting the burst-enable

Cache Mapping Techniques

One of the key aspects of caching is the method by which main memory is mapped to the cache. The method is very important because the cache is typically much smaller than main memory. The result, of course, is that several lines in main memory may map to a single cache line. The objective is to maximize cache hits and to minimize thrashing (repeated cache misses resulting from consecutive memory accesses mapping to the same cache line). The cache needs to keep an inventory of its contents so that it can react the next time the processor requests data. The high order portion of the requested address (the tag) is compared to this list of information held in the cache. To indicate a match, logic compares each tag to the appropriate bits from the requested address. The amount of comparison logic depends on the cache's mapping technique. The most frequently used mapping techniques (also known as placement policies) are direct, fully associative, and set associative.

Direct mapping is the simplest of the placement policies. Any given line from main memory is placed in the cache at the same line modulo N (where N is the number of lines in the cache). The real address is broken into three fields: the tag field, the line field, and the byte field. The tag field checks whether the addressed entry in the cache contains the requested line. The line field is used to access an entry in the cache. The byte field is used to address bytes within a line. Replacing lines in the cache (replacement policy) is trivial. Because a particular line in memory maps to a particular cache line, there is no choice as to which cache line is replaced.

The *fully associative mapping* technique represents the opposite extreme from direct mapping. Any line from main memory may be mapped to any entry in the cache. A large tag field is required (enough to address each line of the cache), because it is possible to map to any line in the cache. The associative comparison of the tags is a time-consuming process since it extends over the entire length of the cache. Because data may be placed anywhere in the cache, additional logic is required to determine where the new data will be placed. This determination is complicated by a full cache (which line should be replaced?). Fully associative caches normally use a replacement policy that saves recently used data and instructions. This mapping method produces very high hit ratios and eliminates thrashing, but at the expense of complicated and costly hardware.

A good compromise between the two extreme cases is called the *set-associative mapping* technique. This method allows main memory lines to be mapped to a limited number of cache entries. Each alternate cache mapping for a given location in memory is termed a way. Therefore, a two-way set-associative cache allows any location in memory to map to two cache entries. Similarly, a four-way set-associative cache allows four entries for any given memory location. Generally speaking, the performance increase from a set-associative cache with more than four ways is overshadowed by the complexity of its implementation. The advantage over full associativity is readily seen. A portion of the address is now used to select a set (the number of sets is the number of lines in a cache divided by ways). The tags are then associatively compared for the number of ways rather than for the number of lines in the cache, greatly reducing the number of associative compares that are necessary. This, of course, makes for a simpler hardware implementation and a faster cache-lookup. While the hit ratio is not as high as for the fully associative method, it is better than for the direct mapped case. □

— JM and TR

bits in the Cache Control Register. The '040 has no such feature for the disable of burst mode.

Another significant difference between the '040 and '030 involves the cache addressing method. The MC68040 caches are physically addressable rather than logically addressable (as in the MC68030). Physical addressing provides some distinct advantages over logical. It is not necessary for the operating system to flush the caches on a task switch. Physical addressing also allows external access to the caches (bus snooping) without the addition of reverse physical-to-logical address translation. Physical addressing into the caches with a 4K minimum page size allows cache lookup to run concurrently with address translation. Cache access concurrent with address translation guarantees a single-cycle cache lookup.

FLOATING-POINT IMPLEMENTATION

The Floating-Point Unit is now on the same chip as the core processor. It has been optimized and can concurrently work with the Integer Unit. Some of the floating-point instructions now show a ten times improvement in speed. Not all of the floating-point instructions, however, were implemented on-chip. Certain instructions are emulated by the '040 and use the floating point instructions on-chip. The actual implementation details are beyond the scope of this article.

While design philosophies and processor architecture have gone through a complete renaissance, the processor remains approachable and compatible. While we have covered the major differences only, we hope to have motivated you to delve deeper into the architecture of the Motorola MC68040. ■

John Meek was involved in the design of mainframe computers at Amdahl and in the design of the HEP II supercomputer. He has worked in Europe as a research engineer and is currently the director of engineering at Progressive Peripherals & Software. Tim Reese has developed a number of accelerated graphics systems for companies in the U.S. and in Europe and was involved in designing Sun-compatible workstations at Solbourne Computers. He is currently the lead hardware engineer at PP&S. Contact them c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or on BIX (r.brothwell).*

References

Bill Ledbetter, Jr., et al., "The 68040 Integer and Floating-Point Units," *Proceedings 1990* COMPCON, (Feb. 26–March 2, 1989), IEEE.

James E. Smith, et al., "Instruction Cache Replacement Policies and Organizations," *IEEE Transactions on Computers*, Vol. C-34, #3, March 1985, Chapter 10, IEEE.

Motorola Inc., *MC68040 32-Bit Microprocessor User's Manual*, Motorola, 1989.

Motorola Inc., *MC68030 32-Bit Microprocessor User's Manual*, Prentice Hall, Englewood Cliffs, N.J., 1990.

A.J. van de Goor, *Computer Architecture and Design*, Addison-Wesley Publishing, 1989. □

ON DISK

Tag Tips

By John Toebes

WITH THE 2.0 operating system, Commodore recognized that they had to extend a number of existing data structures to support new features. In doing so, they encountered a number of difficulties:

- There was no room in the existing structures.
- Defining default values became complicated.
- Adding all the features and extra expansion could waste a lot of space.
- They didn't want to run into this problem again when they added more features.

The first problem was the hardest to resolve. There was no room in such structures as Intuition's NewWindow for the likes of 3-D-look colors, public screen information, and zoom gadget requests. It meant that existing programs could not request any of these new fancy features unless they were willing to break compatibility—not considered to be a good idea.

There are many tricks to make structures upward compatible by stealing bits or putting illegal values in certain fields so the program knows it has new-style structure, but you can go only so far. The process soon becomes too complex to understand and distinguish. By attempting to leave a lot of room in a structure for expansion, you have to rely on determining appropriate default values that will be acceptable in the future. This is not only wasteful of space, but requires being absolutely certain of what the field may be used for to determine an unused value to stick in it.

To combat this problem, Commodore adopted the concept of *tags*. To understand how they work and solve the problem, let's look at a structure conceptually. For each field in a structure, you have a name that identifies the field and a value that you can extract from it. Simply put, structure->member will have a given value and the compiler must make the match-up between the name you use and the value that it extracts. Because the compiler has assigned offsets (and eventually storage) within the structure for each field, the slot must always contain a value. If you want the field to be optional, you must either come up with a null value to indicate so or use some bits elsewhere to indicate that it should not be used.

THE STRUCTURE OF TAGS

With tags, this mapping from a name to the (potential) value is performed not by the compiler at compile time, but by a series of run-time routines. The concept of using a name to identify the field still applies, but instead of the name indicating a storage location, it is used to scan a table for any occurrence of a match. Think of the actual implementation of this table as a array of name and value pairs. For simplicity, all values are stored as 32-bit values. This turns out to be convenient on the Amiga, as almost everything can be represented in 32 bits (the only exceptions are double-precison floating-point values). You will find the following definition for a TagItem in utility/tagitem.h:

```
struct TagItem {
Tag   ti_Tag;   /* a ULONG type as we will see later */
ULONG  ti_Data;
};
```

A typical tag list looks like:

TAG	Data
TOP	10
WIDTH	100
HEIGHT	50
LEFT	10
TAG_DONE	<Don't Care>

The tag list above takes up 40 bytes in memory. The first thing to understand is that there is no particular order implied about the entries (except for TAG_DONE). The tag list consisting of:

TAG	Data
LEFT	10
WIDTH	100
HEIGHT	50
TOP	10
TAG_DONE	<Don't Care>

is identical to the first list for all intents and purposes. To interpret a tag list and extract the data from it, you must search from the start of the list until you either encounter a TAG_DONE or the item you desire. What we haven't considered is how this magic tag is represented. Under X-Windows, the TAG is actually a pointer to a string that contains the text LEFT, WIDTH, or whatever the tag item is. While this allows for a lot of flexibility, it takes quite a bit of optimization to

make it efficient to access the tag lists. Instead, Commodore uses defined numeric values for each of the tag values. A value of 0 or TAG_DONE indicates the end of the list. (We will examine a few other special values soon.)

The tags of LEFT, WIDTH, HEIGHT, and TOP are what are called *user tags*. They always have the high-order bit set (0x8000000) to distinguish them from *system tags*. To assist in ensuring that they can be identified as user tags, you should always define them relative to the system-defined TAG_USER, which is set to 0x8000000. The file utility/tagitem.h also defines a data-type tag that is a ULONG and is the appropriate type with which to declare any tags. Typical C code defines tags as follows:

```
#define LEFT  TAG_USER+1
#define TOP  TAG_USER+2
#define HEIGHT TAG_USER+3
#define WIDTH TAG_USER+4
```

So, when you want to find the WIDTH from the tag list, you search for an entry that has the value 0x8000004 in the TAG field. Then you use the corresponding DATA field. This is actually quite simple and can even be accomplished in C as:

```
ULONG FindTag(Tag Tagval, ULONG defval, struct TagItem *taglist)
{
   while(taglist->ti_Tag != TAG_DONE)
   {
      if (taglist->ti_Tag == Tagval)
         return(taglist->ti_Data);
      taglist++; /* advance to the next tag */
   }
   return(defval); /* We didn't find it, return the default value */
}
```

In utility.library, Commodore provides a more advanced version of this routine called GetTagData(). It takes the same parameters, but it also understands the other system tags we will see later.

A couple of other useful routines you will find in utility.library are:

Tagitem = FindTagItem(Tag, Taglist);: Locates the TagItem entry in a Taglist that corresponds to the given tag. If it can't find the Tag in the list, it returns NULL.

flag = TagInArray(Tag, TagList);: Returns TRUE if the tag exists in the TagList and FALSE otherwise. It is very useful for determining if a particular feature is being requested.

As you can see, even though the TagItem data is a ULONG, you can store BOOLs, shorts, and even pointers in the entries. You can even have entries where the mere presence in the array is enough to trigger a feature.

SYSTEM TAG VALUES

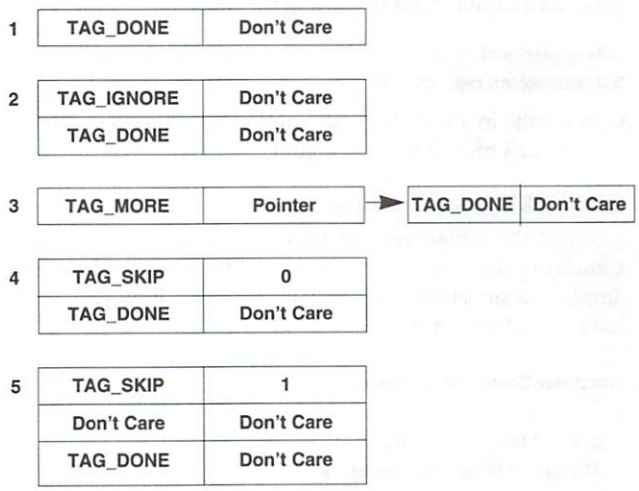
In addition to the TAG_DONE system tag, there are three other system tags:

TAG_IGNORE: This is simply a place holder to tell the tag-list processing to ignore this entry and the data that goes with it. This is useful for hiding an entry in the array.

TAG_MORE: The data for this entry is a pointer to another tag list that is considered to be a continuation of the current tag list. Note that any entries after TAG_MORE in the current list are ignored. You can think of this system tag as a form of goto.

TAG_SKIP: The data for this entry is the number of TagItem entries following the current entry that are to be ignored. A TAG_SKIP 0 is the same as a TAG_IGNORE. Think of this as a jump-ahead-a-few-entries selection.

A little creative work can build tag lists like:



All of these are equivalent examples of empty tag lists. The first is just one entry that says it is the last entry. The second list has a single entry, but it is a TAG_IGNORE that the tag-list routines, of course, ignore. The third one is an example of a TAG_MORE that points to another tag list that happens to be an empty list. (*Never* point a TAG_MORE back at itself. You might get tired of waiting for the tag-list routines to never complete the traversal.) Example four uses TAG_ ►

SKIP to ignore the current entry. The difference between this and the TAG_IGNORE is that the data field for the TAG_SKIP must be 0 and the TAG_IGNORE data field can be set to anything. The last case shows how to skip over a subsequent entry without having to rewrite the entire list.

TWO TYPES OF CALLS

With a clear idea of how tags sit in memory, you are ready to learn how to use them with the system routines. There are two ways to call any of the system routines that take tag lists as parameters. You can specify the tag-list entries as parameters to the function and call it through a stub routine in amiga.lib, which turns the parameters into a tag list for you. The other method is to build a tag list ahead of time and then pass the address of the tag list as a parameter to the routine. For example:

```
struct TagItem tagarray[1] = { { TAG_DONE, 0 } };
OpenWindowTagList(nw, tagarray);
```

and

```
OpenWindowTags(nw, TAG_DONE, 0);
```

are functionally identical. The choice of usage is up to you, but you should be aware of some important problems:

- When using the tags as individual parameters, you are going through a stub routine that may prevent you from making the code resident.
- Pushing a lot of parameters on the stack at run time is expensive and slow.
- Some versions of the commercial compilers have problems with such constructs as:

```
struct TagItem tagarray[1] = { { W_NAME, (ULONG)"TitleName" } };
```

Contact your compiler vendor for a solution if necessary.

Unfortunately, there is no consistent naming convention for the tag-list routines. In most cases, the tag-list parameter version has a capital A on the end of the name. For example:

```
SetAttrsA(object, taglist);
SetAttrs(object, tag, val, ... );
```

Consult the include files and AutoDocs for the function you want to call to find the right name.

CONVERTING TO TAGS

All of the advantages of tags are irrelevant if you can't translate your favorite routines to use them. Consider the example of a simple OpenScreen() call. Here's how you would have coded it before 2.0:

```
struct NewScreen NewScreen =
{
   0, 0,   /* Left and top edges   */
   320,200,  /* Width and Height   */
   2,  /* Depth     */
   0, 1,   /* Detail and Block Pens  */
   NULL,   /* Display Modes   */
   CUSTOMSCREEN,  /* Screen Type     */
   NULL,   /* No special font   */
   "My Screen",  /* Screen title    */
   NULL,   /* No screen Gadgets   */
   NULL    /* No custom Bitmap   */
};

... lots of code here ...

screen = OpenScreen(&NewScreen);
```

With the 2.0 OpenScreenTags(), you can code it as:

```
screen = OpenScreenTags( NULL,
SA_Width,  320,
SA_Height, 200,
SA_Depth,  2,
SA_Title,  (ULONG)"My Screen",
SA_Type,  CUSTOMSCREEN,
TAG_DONE,  0);
```

You obtain the values for the tags by reading intuition/screens.h (or the AutoDocs for OpenScreenTagList()). You must know what the tag value names are to use, in order for the system routines to understand what you are asking for. Note that you do not have to specify a BitMap, display modes, or even the left and top edges, because Intuition will use an appropriate default when it does not find an appropriate entry in the tag list.

Another useful tag-list function is System(). While it used to be difficult to execute a command and gain control of its output and how it ran, with System() and a few tag-list entries, you can do almost anything. The include file dos/dostags.h defines everything you can ask of DOS.

TAKE A SECOND LOOK

Now that you have seen a bit of how simple tag usage is, note that you must be careful which tag values you are using. Because each library in the system restarts numbering its tags at TAG_USER, you can encounter many situations in which the same value means different things. For example, a tag value of 0x80000021 indicates SYS_INPUT to dos.library tag routines, while it means SA_LEFT to Intuition routines. A mistaken value can be very hard to debug, so be careful when coding.

The other thing to think about is interpreting the data for the tag items. For some tag values, the data is simply a boolean value (such as SYS_INPUT), while for others it is a numeric value (such as SA_LEFT). You have tags that are for pointers (SA_TITLE) and even those that are for bit flags (SA_Type). Because the compiler does no type checking on these tag lists, you should take care to match the right type of parameter to the tag.

ADVANCED TAG MANIPULATION

Utility.library provides several additional functions for manipulating tag lists. A couple—such as NextTagItem() and CloneTagItems()—you may never use unless you get very heavily into manipulating tags, but the following three can make life very easy:

```
PackBoolTags(flags, TagList, BoolTagMap);
FilterTagList(TagList, TagArray, LOGIC);
MapTags(TagList, MapList, IncludeMissFlag);
```

PackBoolTags() allows you to construct a bitmask of data values based on both a user tag list and predefined defaults. Let's take an example of a window system and a few tags:

```
#define T_BORDER  TAG_USER+1
#define T_SIZING  TAG_USER+2
#define T_CLOSE  TAG_USER+3
```

```
#define T_DRAGBAR TAG_USER+4
#define T_ZOOMBOX TAG_USER+5
#define T_DEPTHBOX TAG_USER+6
#define T_TITLE  TAG_USER+7
```

You can define a correspondence for these bitmasks with (you guessed it) a TagArray:

```
struct TagItem maskmap[ ] = {
   { T_BORDER, 0x0001 },
   { T_SIZING,  0x0002 },
   { T_CLOSE, 0x0004 },
   { T_DRAGBAR,  0x0008 },
   { T_ZOOMBOX,  0x0010 },
   { T_DEPTHBOX,  0x0020 },
   { TAG_DONE,  0  } };
```

Now, if you implement a routine similar to an OpenWindow() routine that takes a tag list to specify the attributes of the window desired, you can use this maskmap tag list to build a mask of the requested features in a single pass. You can also start out with a default set of features (such as a DRAGBAR and a SIZING gadget) and allow the user to disable it if desired. If the routine were called with a tag list such as:

```
struct TagItem example[ ] = {
   { T_BORDER, TRUE },
   { T_DRAGBAR,  FALSE },
   { T_TITLE,  (ULONG)"Title" },
   { T_CLOSE,  TRUE },
   { T_DEPTHBOX,  FALSE },
   { TAG_DONE,  0  } };
```

and you call:

```
final = PackBoolTags( 0x000A, example, maskmap);
```

final is set to a value of 0x0007 requesting a BORDER, SIZING and CLOSE box, but not a DRAGBAR, ZOOMBOX, or DEPTHBOX. To see, take a look at how PackBoolTags works:

1. It starts with the 0x000A passed in (T_SIZING | T_DRAGBAR).

2. It goes to the first entry in example and finds T_BORDER. It searches maskmap and finds it has a value of 0x0001. Because the data for T_BORDER in the example is TRUE, it combines 0x0001 into 0x000A via OR, resulting in 0x000B (T_SIZING | T_BORDER | T_DRAGBAR).

3. It proceeds to the next entry, T_DRAGBAR. Looking it up in maskmap produces 0x0008. Because the example specifies it as false, it combines the current value with the complement via AND and produces 0x0003 (T_BORDER+T_SIZING).

4. It steps to the next entry, T_TITLE. Because it does not find T_TITLE in maskmap, it just ignores T_TITLE and keeps the value of 0x0003.

5. The next entry is T_CLOSE. Maskmap gives it 0x0004, which is added in with OR, because the example specified it as TRUE. Our value is now 0x0007 (T_BORDER | T_SIZING | T_CLOSE).

6. Next, maskmap yields 0x0020 for T_DEPTHBOX. The complement of 0x0020 is combined with 0x0007 via AND, because the example specified it as FALSE. Because the bit was not on to begin with, the value remains the same.

7. The last entry is TAG_DONE. PackBoolTags() has done its work so it quits. The extremely nice aspect of this interface is that you can change the defaults and even the internal bit ordering without having to change a single line of user code. The code continues to request features and the target routine can optimize it into its own bit fields.

The other routines, FilterTagList() and MapTags(), are useful for changing tag lists around when moving from one set of tag values to another. FilterTagList() changes all located tags to a TAG_IGNORE. For example, if you want to eliminate all SC_VAL1 and SC_VAL2 tag entries from a tag list use:

```
Tag filterlist[] = { SC_VAL1, SC_VAL2, TAG_DONE };
FilterTagList(taglist, filterlist, TAGFILTER_AND);
```

Note that although filterlist is terminated by a TAG_DONE, it is not a tag list. It is simply an array of tag values terminated by a TAG_DONE. After the operation, all occurrences of SC_VAL1 and SC_VAL2 entries in the list will be set to TAG_IGNORE. You can do the opposite—filter out everything but the ones specified with:

```
FilterTagList(taglist, filterlist, TAGFILTER_NOT);
```

For a little more sophisticated filtering of tags, you can use MapTags(). Like FilterTagList(), it applies mass changes to a list, but instead of putting in TAG_IGNORE, it allows you to replace the tags with anything. For example, consider:

```
struct TagItem list[ ] = {
   { MY_SIZE, 71 },
   { MY_WEIGHT,  200 },
   { TAG_END, 0  } };
```

and a mapping tag list of

```
struct TagItem map[ ] = {
   { MY_SIZE, HIS_TALL },
   { TAG_END, 0 } };
```

where MY_SIZE might be a form of height in my set of routines, while another set of routines could expect a HIS_TALL tag for the same information. Instead of making the user replicate the information in the tag list, you can change it on the fly. If you call MapTags(list, map, 0), list becomes:

```
{ HIS_TALL,  71 }
{ TAG_IGNORE, 200 }
{ TAG_END,  0 }
```

If, instead, you call MapTags(list, map, 1), it leaves MY_WEIGHT in to give you:

```
{ HIS_TALL,  71 }
{ MY_WEIGHT,  200 }
{ TAG_END,  0 }
```

Note that, for safety reasons, attempting to map some tag value to TAG_END will result in the MapTags() routine substituting TAG_IGNORE.

WRAPPING IT UP

Tags can be very powerful and can make your code quite a bit simpler in the long run. In particular, the introduction of tags ensures that your code will continue to function with newer releases of the operating system that add more features, while still allowing for expansion. As long as you keep track of the tag type and the type of values they take, you should have little trouble using them. ■

John Toebes is coordinator for The Software Distillery. He was a major developer of the SAS/C system for AmigaDOS. Contact him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or on CompuServe (72230,303).*

ON DISK

2.0's Graphics.Library

By Spencer Shanson

THE SURFACE DIFFERENCES between 2.0 and previous versions of the operating system jump at you the moment you boot your Amiga. Gone are the old blue-orange-black-white color scheme and flat design; in their places are refreshing greys and blues, drop shadows, new gadgets, and a generally much more professional look. Beneath the Intuition and Workbench facelift lie the more exciting differences—the changes to the graphics.library, which houses the code that makes all the colors, displays, lines, and icons available, shoves it onto your screen, and no sooner than one screen is done makes the next 1/60 of a second later. Although average users are sheltered from it, programmers will appreciate how much work has gone into the graphics.library for 2.0. (Note that I will refer to 2.0 as V37 from now on, in line with 1.3 being V35, and so on. V36 was the alpha and early beta version of 2.0. When opening a system library, you should use V36 if you know a feature is available in V36, otherwise use V37. There are very few features in V37 that were not in V36.)

CHIPS (BUT NO FISH)

The main purpose of the 2.0 graphics.library was to provide support for Commodore's Enhanced Chip Set (ECS). The ECS consists of new versions of the display chip (Denise) and the graphics and DMA engine (Agnus). What does the ECS give you that the previous chip revisions did not?

Agnus can now work in PAL or NTSC mode. A jumper on the A2000 or a track on an A500 configures your machine to be PAL or NTSC by default, but this can be overriden by the V37 software. The ECS also provides a new display mode called SuperHires, which gives twice the resolution of a hires screen (1280 pixels wide standard). Because of DMA restrictions, however, you can run this mode with a maximum of only two bitplanes (four colors). As well as providing timings for NTSC/PAL and SuperHires, the ECS Denise can also be programmed with variable beam rates, meaning Denise can provide a VGA-like mode (640×480, noninterlaced) called Productivity mode on a multisync monitor. Variable beam rates also need ECS Agnus for DMA timing. Again, DMA restricts Productivity mode to two bitplanes.

All this is very nice, but ECS Agnus also takes a positive step towards alleviating the regular gripe of "only having 512K of chip RAM," because it can support one megabyte of chip RAM (or two megabytes on the A3000). You can now have larger pictures, animations, and sound samples in that precious chip RAM of yours. To utilize the extended memory, Agnus has increased its blitting range from 1008×1024 pixels to 32K×32K pixels! As a given, this also extends its line-drawing capability to lines that are 32K pixels long.

Denise is responsible for defining the display window on your monitor. (The display window is the area on a screen in which all the action happens.) Denise's job is to take the display data from Agnus, mix it with the sprite data, prioritize the sprites and playfields, convert palette numbers to Red, Green, and Blue values, and shove the result out of the RGB port to your monitor. If that's not enough, it also provides genlock support.

Under 1.3 and earlier, Denise was restricted in display-window size. Horizontally, the display window had to start in the left 3/4 of the display and finish in the right 1/4. Vertically, the display had to start in the upper 2/3 and finish in the lower 1/2. These restrictions were imposed because the required Most Significant Bits were nonprogrammable, and were hardcoded in silicon. In the ECS, a new programmable register was added to supersede the hardcoded values—so *poof* go those restrictions!

Probably the most interesting features of the ECS Denise, especially for the video world, are its new genlock modes. The old Denise set a pin on the RGB port called ZD (Zero Detect) when the chip came across a pixel that was set to color 0 (the background color). An external genlock could watch this pin, and when the pin was asserted, it could output a video signal (from a VCR, video camera, or whatever) instead of the Amiga's regular signal. That's how genlocks work. But just genlocking on the background color was not always enough for some people. They wanted to be able to genlock on any color on the screen (called chromakeying), and some more expensive genlocks provided this feature themselves. The ECS Denise can now do that as well, plus it provides bitplane overlay, so an entire bitplane controls genlocking (to provide those keyhole-type effects), and BorderBlanking creates a transparent "frame" surrounding the active area.

THE GRAPHIC DATABASE

"So," you say, "that's what the ECS hardware does. How do I make my application software use these new functions?"

As before, you can determine the amount of chip RAM you have available by calling:

AvailMem(MEMF_CHIP)

Your code should be able to handle the (possible) new size.

The blit sizes are a different problem. Following the tradition of assumed values ("blits are always 1K×1K"), we could easily suppose "if a machine has an ECS Agnus, we can blit 32K×32K." We could, but we shouldn't. What if Commodore releases another version of Agnus that supports even larger

"V37 graphics.library has a graphics database that contains information on dimensions, monitors, and display for every known display mode."

blits. Are you supposed to use another set of hardcoded values? To alleviate this problem (and many others), V37 graphics.library has a graphics database that contains information on dimensions, monitors, and display for every known display mode.

Prior to V37, you set display modes in the ViewPort->Modes field, which was 16 bits wide (one WORD). This field was getting fairly cramped, especially with the need for the new modes (SUPERHIRES) and more monitor types (PAL, NTSC, VGA). Therefore, every display mode in V37 has an associated 32-bit (one LONG) DisplayID value. The upper WORD holds the monitor type number, while the lower WORD defines the monitor's various modes. For compatibility with existing software, if the upper word of the LONG DisplayID is 0, then the "default" monitor (NTSC or PAL, depending on your Amiga's configuration) is assumed. The monitor numbers and the DisplayID values are defined in a new header file, graphics/displayinfo.h.

You can find the DisplayID of a ViewPort using the new V37 graphics function:

```
ULONG GetVPModeID(struct ViewPort *vp).
```

To be compatible with future revisions of the OS, you should use this function from now on, rather than using ViewPort->Modes. In addition, be sure to check the return value against INVALID_ID (in graphics/displayinfo.h) for errors.

When you have this ID, you can use it to read information from the database. Four types of information are available—Monitor, Dimension, Name, and Display—and each contains information similar to the following sample (which is for the NTSC hi-res interlaced mode):

```
*** Mode 0x00019004 ***
Name : NTSC:Hires-Interlaced

MonitorInfo
{
ViewPosition        = (0x73, 0x2C)     = (115, 44)
ViewResolution      = (0x2C, 0x34)     = (44, 52) (ticks per pixel)
ViewPositionRange Rectangle (fixed, hardware dependent) =
   (0x5D, 0x15) – (0x88, 0x3F)         =
   (93, 21) – (136, 63)
TotalRows           = 0x106 = 262
TotalColorClocks  = 0xE2 = 226
MinRow              = 0x15 = 21
Compatibility       = 0x0 = MCOMPAT_MIXED
(can share display with other
MCOMPAT_MIXED modes)
}

DimensionInfo
{
MaxDepth = 0x4  /* The number of bitplanes supported */
MinRasterWidth = 0x20 = 32 pixels
MinRasterHeight = 0x1 = 1 pixels
MaxRasterWidth = 0x3FF0 = 16368 pixels /* determined by Agnus type */
MaxRasterHeight = 0x4000 = 16384 pixels  /* determined by Agnus type */
Nominal Rectangle (Standard Dimensions) =
   (0x0, 0x0) – (0x27F, 0x18F) =
   (0, 0) – (639, 399)
MaxOScan Rectangle (fixed, hardware dependent) =
   (0xFFFFFFD4, 0xFFFFFFD2) – (0x2A7, 0x1B3) =
   (-44, –46) – (679, 435)
VideoOScan Rectangle (fixed, hardware dependent) =
   (0xFFFFFFD4, 0xFFFFFFD2) – (0x2B3, 0x1B3) =
   (-44, –46) – (691, 435)
TxtOScan Rectangle (Editable via Preferences) =
   (0x0, 0x0) – (0x2A7, 0x18F) =
   (0, 0) – (679, 399)
StdOScan Rectangle (Editable via Preferences) =
(0xFFFFFFF8, 0x0) – (0x2A7, 0x18F) =
   (-8, 0) – (679, 399)
}

DisplayInfo
{
NotAvailable = 0x0  /* if non-zero, this mode may not be
                     * available if, for example, this mode
                     * requires ECS chips, and the Amiga
                     * does not have them. You can determine
                     * a mode's availability with the V37
                     * ModeNotAvailable() function
                     */
PropertyFlags   = 0xBC1
   LACE SPRITES GENLOCK WB DRAGGABLE BEAMSYNC
Resolution          = (0x16, 0x1A) = (22, 26) (ticks per pixel)
PixelSpeed          = 0x46ns = 70ns
NumStdSprites   = 8
PaletteRange        = 0x1000 = 4096
SpriteResolution = (0x2C, 0x34) = (44, 52) (sprite ticks per pixel)
}
```

As you can see, the information for each possible mode is fairly extensive. (Note: V37 now provides OS support for overscan.) You can get the database via a DisplayInfoHandle. ►

First, find the handle:

```
handle = FindDisplayInfo(ULONG DisplayID).
```

Now, pass it to:

```
ULONG GetDisplayInfoData(DisplayInfoHandle handle, UBYTE *buf,
   ULONG size, ULONG tagID, [ULONG id]).
```

This takes the handle, a pointer to a buffer, the number of bytes to copy from the database into your buffer, and an identifier for the type of information you need. Alternatively, you can pass NULL as the handle with a DisplayID value. To iterate through the entire database of known modes use:

```
ULONG NextDisplayId(ULONG DisplayID)
```

As an example of how all this works together, the following code prints the maximum depth (bitplanes) supported by each known display mode:

```
#include <graphics/gfxbase.h>
#include <graphics/displayinfo.h>

struct GfxBase *GfxBase;

void main()
{
   APTR buf;
   ULONG ID = INVALID_ID;

   if (GfxBase = OpenLibrary("graphics.library", 36))
   /* V36 is the early KS2.0 */
   {
      if (buf = AllocMem(MAX(sizeof(struct DimensionInfo), sizeof
         (struct NameInfo)), MEMF_CLEAR))
      {
         while ((ID = NextDisplayInfo(ID)) != INVALID_ID)
         {
            /* iterate through each known ID */

            printf("ID 0x%lx ", ID);
            if (GetDisplayInfoData(NULL, buf, sizeof(struct NameInfo),
               DTAG_NAME, ID))
            {
            printf("Name – %s, ", ((struct NameInfo *)buf)->Name);
            }
            if (GetDisplayInfoData(NULL, buf, sizeof(struct Dimension-
               Info), DTAG_DIMS, ID))
            {
               printf("MaxDepth = %ld\n", ((struct DimensionInfo *)
               buf)->MaxDepth);
            }
         }
         FreeMem(buf, MAX(sizeof(struct DimensionInfo), sizeof(struct
         NameInfo)));
      }
      CloseLibrary(GfxBase);
   }
}
```

GENLOCKS AND KEYHOLES

All the new genlock information is stored in a ColorMap structure, and can be controlled via the new VideoControl() function. Because the ColorMap has grown and future versions are also likely to grow, you should be getting and freeing your ColorMap structure with the GetColorMap() and FreeColorMap() functions. These graphics functions know exactly how big the ColorMap structure is for the given Kickstart version number. Therefore, you are guaranteed to have a valid up-to-date ColorMap structure. A ColorMap is associated with a ViewPort, so you can use VideoControl() to alter the genlock mode on a ViewPort-by-ViewPort basis.

VideoControl() takes a ColorMap structure and a pointer to a TagList:

```
ULONG VideoControl( struct ColorMap *cm, struct TagItem *ti);
```

The TagList is a list of instructions for VideoControl() to perform on the ColorMap. Some of these instructions require extra data, and some are Boolean. After receiving the TagList, VideoControl() can alter it. As a demonstration, the following routine determines if the genlock BorderBlank feature is enabled for the current ColorMap, and enables ChromaKey for color 3. All the tags are defined in graphics/videocontrol.h. (For a general discussion of tags, see "Digging Deep in the OS," p. 8.)

```
ULONG GenlockStuff(struct ColorMap *cm)
{
   #define TAG_COUNT 3   /* # of instructions to pass to VideoControl() */
   struct TagItem *ti;
   ULONG result = 1;

   if (ti = (struct TagItem*)AllocMem((sizeof(struct TagItem) *
   TAG_COUNT), 0))
      {
      ti[0].ti_Tag = VTAG_BORDERBLANK_GET; ti[0].ti_Data = NULL;
      ti[1].ti_Tag = VTAG_CHROMA_PEN_SET; ti[1].ti_Data = 3;
      ti[2].ti_Tag = VTAG_END_CM; ti[2].ti_Data = NULL;
      /* shows end of TagList */

      /* now the TagList is set up, pass it to VideoControl(). A non-
      NULL return
      * value signifies an error – either a bad ColorMap
      * (of type pre-V36), of a bad taglist
      *
      * ti[0].Tag will be changed to either VTAG_BORDERBLANK_SET
      * or
      * VTAG_BORDERBLANK_CLR, depending on its setting.
      */

      if ((result = VideoControl(cm, ti)) == NULL)
      {
            printf("BorderBlank is %s\n", (ti[0].ti_Tag == VTAG_BOR-
            DERBLANK_SET) ? "Set" : "Clear");
      }
      else
      {
         printf("VideoControl() error\n");
      }
      FreeMem(ti, sizeof(struct TagItem) * TAG_COUNT);
   }
   else
   {
      printf("Could not allocate memory\n");
   }

      return(result);
}
```

You can control other aspects of the ColorMap (and hence the ViewPort) with VideoControl(), as well. For example, as anyone who has ever used UserCopperLists knows, UserCopperLists "leak" through into other screens as the screens are dragged around. Look at PhotonPaint for an example: Put PhotonPaint in back of all the other screens, and then drag one of the frontmost screens down, so that the top of the PhotonPaint screen shows. With VideoControl(), you can now turn on "UserCopperList clipping" and stop the leaking.

OVERSCAN—WALL-TO-WALL VIDEO

You can also set a ViewPort's DisplayClip with VideoControl(). A DisplayClip is the total area of a ViewPort and can define regions greater than the standard area. The effect of this capability is known as overscan. Overscan is used by many applications, notably paint packages, to increase the working area. It is also useful when sending the output of your Amiga to videotape to reduce the blank area around the View, giving that wall-to-wall effect. Before V37, however, applications provided overscan through a variety of unfriendly tricks, because the OS offered no direct support. Some of those tricks still work under V37, but there is no excuse for using them anymore. Make it easy on yourself and other programmers—play by the 2.0 rules.

To allow for the DisplayClip, the ViewPort structure had to be extended. This was no mean trick. The ViewPort cannot be physically increased in size because Intuition has an instance of a ViewPort structure in its Screen structure—extending it would mean changing the offsets of much data in the Screen and would break just about every piece of software ever published for the Amiga! So, a ViewPortExtra structure was created and is "magically" linked to its associated ViewPort by the OS.

"The DimensionInfo has five types of rectangles defined: Nominal, MaxOScan, VideoOScan, TxtOScan, and StdOScan."

To get a ViewPortExtra structure, call the V37 function GfxNew():

```
struct ExtendedNode *GfxNew(ULONG node_type);
```

For example:

```
struct ViewPortExtra *vpe = (struct ViewPortExtra *)GfxNew(VIEW-
PORT_EXTRA_TYPE);
```

The ViewPortExtra structure (like ViewExtra) is headed by an ExtendedNode, hence the type cast. These are defined in graphics/gfxnodes.h.

Because ViewPortExtra may grow in the future, you must use the GfxNew() function to allocate one of these structures to ensure that your software is compatible with future OS versions. When you finish with the structure, you should return the memory used to the system with GfxFree():

```
void GfxFree(struct ExtendedNode *en);
```

So, now that you have a ViewPortExtra structure, you can define a DisplayClip. This is simply a rectangle, defined by the top-left and bottom-right corners, using the units of the ViewPort's mode. In other words, the units are in LORES pixels for a LORES ViewPort, HIRES pixels for a HIRES ViewPort, and so on. The origin of the rectangle is the position of graphics' View.

If you take another look at the features of the database, you will see that DimensionInfo has five types of rectangles defined: Nominal, MaxOScan, VideoOScan, TxtOScan, and StdOScan. Nominal is the standard DisplayClip for this mode, such as 320x200 for a LORES noninterlaced ViewPort. MaxOScan is the maximum DisplayClip the software will handle, while VideoOScan is the absolute maximum the hardware can handle. TxtOScan is the DisplayClip in which all text rendered will be visible. StdOScan is the region that extends to the bezel of your monitor. You can alter both TxtOScan and StdOScan from the Overscan preferences editor on Workbench 2.0.

If you want your application to open with the user's TxtOScan for a HIRES ViewPort, first query you the database. Then copy the TxtOScan rectangle that was copied into the buffer you passed in order to GetDisplayInfoData() into the ViewPortExtra->DisplayClip rectangle. Next, you associate this ViewPortExtra with its ViewPort, and repeat the process using VideoControl() with the VTAG_VIEWPORTEXTRA_SET tag. Note that you do not have to use any of the defined DisplayClips; you can create any you like, but anything larger than MaxOScan is perilous!

COPPER? I HARDLY EVEN KNOW 'ER

The Copper (coprocessor) is a simple processor that understands only three instructions: MOVE, WAIT, and SKIP. It is able to WAIT until the video beam that sweeps across your display has reached at least a certain position, and then MOVE data into the custom chip registers that make the Amiga different from all other computers, such as the bitplane pointers and registers that control the display mode. It is this processor that allows the Amiga to show different display modes on-screen at the same time (when you slide screens around). Just like any other processor, it executes a list of instructions.

Many games use their own Copper lists for display tricks, but some like to "borrow" and corrupt graphics' Copper lists. Most of the time they get away with this, but there are some games that make mistaken assumptions about graphics' Copper lists. The authors of these programs assumed that, because the format of the Copper lists did not change between early versions of the OS, the lists would not change in the future. *They were wrong!*

The format of the Copper lists has changed considerably between V35 and V37. Software that expects certain Copper instructions always to be at certain offsets from the start of the list breaks under V37. Other programs assume that the Copper lists always load some chip registers with the same data value and therefore never load those registers. These programs also break.

Do not assume anything about the Copper lists. They have changed in the past and are likely to change in the future. Note well the comment in graphics/copper.h: ►

```
/* private graphics data structure */
```

If you need to play with the system's Copper lists, install your own UserCopperLists using the graphics.library macros—CINIT, CMOVE, CWAIT, and CEND—defined in graphics/gfxmacros.h. Examples of their use are the Sliced-HAM (SHAM) and DynamicHiRes tricks that allow more colors on the screen simultaneously by changing the color palette on every line.

TEXT (VERY FITTING!)

Some new functions were added to graphics.library's Text module to help in fitting text into a defined area. Bear in mind that, under V37, the default font can be of any size (as set by the text preferences); you cannot assume anything about the font your application will be rendering in, unless you specify a particular font in your code.

The problem with the old TextLength() function was that the value it returned was not the number of pixels with which the string was rendered, but the value that was added to the RastPort->cp_x value. If, for example, you rendered the text "Hello" into a rastport, and you later wanted to delete that text, you could have used TextLength() to determine the number of pixels in the string and deleted the text with that result.

If the text was rendered in italics, however, the value returned from TextLength() (which is the value added to RastPort->cp_x) would have been shorter than the actual number of pixels used to render the string. V37's new function, TextExtent(), finds the "bounding box" of a text string, given the font's size and text attribute (bold, italic, underlined, and so on):

```
void TextExtent(struct RastPort *rp, STRPTR string, WORD count,
struct TextExtent *te);
```

This function takes a RastPort, a pointer to a string, a count of the number of characters in the string, and a pointer to a TextExtent structure, which will be filled by the function for the result. The new TextExtent structure is defined in graphics/text.h, and is filled in as follows (from Commodore's AutoDocs):

```
te_Width—Same as TextLength() result: the rp_cp_x advance that
rendering this text would cause.
te_Height—Same as tf_YSize. The height of the font.
te_Extent.MinX—The offset to the left side of the rectangle this would
render into. Often 0.
te_Extent.MinY—Same as –tf_Baseline. The offset from the baseline to
the top of the rectangle this would render into.
te_Extent.MaxX—The offset of the right side of the rectangle this would
render into. Often the same as te_Width-1.
te_Extent.MaxY—Same as tf_YSize-tf_Baseline-1. The offset from the
baseline to the bottom of the rectanangle this would render into.
```

Another feature missing in previous versions of the OS is a function to determine how many characters of a string would fit wholly in a defined area. The TextFit() routine cures that problem in V37. This function also fills a TextExtent structure:

```
ULONG TextFit(struct RastPort *rp, STRPTR string, UWORD strlength,
    struct TextExtent *te, struct TextExtent *constraining_te,
    WORD strdirection, UWORD constraining_BitWidth,
    UWORD constraining_BitHeight);
```

Constraining_te is the text extent that the text must fit in. If this is NULL, then constraining_BitWidth/Height defines the text extent instead. Strdirection can be either 1 or –1. If 1, the string is anchored at the left side of the box. If –1, the string is anchored at the right side of the box, and the pointer to the string should point to the string's last character.

The following crude fragment of code prints the first n letters of the alphabet that will fit in a box that is 200 pixels wide by 100 pixels high, given the rastport's font and style. Note that if the font is taller than 100 pixels, then no characters will fit in the box.

```
void alphabet(struct RastPort *rp)
{
#define STRING "abcdefghijklmnopqrstuvwxyz"
#define LENGTH strlen(STRING)
#define WIDTH 200
#define HEIGHT 100

    struct TextExtent *te;
    ULONG count;

    if (te = AllocMem(sizeof(struct TextExtent), MEMF_CLEAR))
    {
        count = TextFit(rp, STRING, LENGTH, te, NULL, 1, WIDTH,
        HEIGHT);
        if (count)
        {
            Move(rp, 0, HEIGHT);
            Text(rp, STRING, count);
        }
        FreeMem(te, sizeof(struct TextExtent));
    }
}
```

FONTS

Before V37, if you tried to open a font and the size you wanted was not available, then the open failed. Under V37 the font size specified will be created (scaled) from the closest available defined size. Therefore, your application should be prepared to handle fonts of any size. The routine that does the actual scaling is called BitMapScale() and can be used by your applications. Look in graphics/scale.h for a description of the structure this function uses. For an example of usage and details on known bugs and limitations, study the sample source code in the Shanson drawer of the accompanying disk.

Finally, the TextFont structure needed to be extended under V37. Because many applications use embedded TextFont structures in their code, such as:

```
struct TextFont myTextFont
```

the structure could not be directly expanded without breaking much existing software. So, whenever you call OpenFont(), the system creates a new TextFontExtension and magically attaches it to the TextFont structure returned. The TextFontExtension is removed when you call CloseFont(). Some applications, however, use their own hardcoded fonts that are never opened with OpenFont(), but the system still creates the TextFontExtension when you call SetFont() for that font.

This creates a problem: The system has allocated memory for the new structure, but has no way of knowing when to return that memory, because the font is never closed. Conse-

Continued on p. 63

SAVE $35.75

A source of technical information for the serious Amiga professional.

Introducing *The AmigaWorld Tech Journal*, the new source to turn to for the advanced technical information you crave.

Whether you're a programmer or a developer of software or hardware, you simply can't find a more useful publication than this. Each big, bi-monthly issue is packed with fresh, authoritative strategies to help you fuel the power of your computing.

Trying to get better results from your BASIC compiler? Looking for good Public Domain programming tools on the networks and bulletin boards? Like to keep current on Commodore's new standards? Want to dig deeper into your operating system and even write your own libraries? Then *The AmigaWorld Tech Journal* is for you!

Our authors are programmers themselves, seasoned professionals who rank among the Amiga community's foremost experts. You'll benefit from their knowledge and insight on C, BASIC, Assembly, Modula-2, ARexx and the operating system—in addition to advanced video, MIDI, speech and lots more.

Sure, other programming publications may include some technical information, but none devote every single page to heavyweight techniques, hard-core tutorials, invaluable reviews, listings and utilities as we do.

Every issue includes a valuable companion disk!

And only *The AmigaWorld Tech Journal* boasts a technical advisory board composed of industry peers. Indeed, our articles undergo a scrupulous editing and screening process. So you can rest assured our contents are not only accurate, but completely up-to-date as well.

PLUS! Each issue comes with a valuable companion disk, including executable code, source code and the required libraries for all our program examples—plus the recommended PD utilities, demos of new commercial tools and other helpful surprises. These disks will save you the time, money and hassle of downloading PD utilities, typing in exhaustive listings, tracking down errors or making phone calls to on-line networks.

The AmigaWorld
TECH JOURNAL

In every issue of *The AmigaWorld Tech Journal*, you'll find...

- Practical hardware and software reviews, including detailed comparisons, benchmark results and specs.
- Step-by-step, high-end tutorials on such topics as porting your work to 2.0, debugging, using SMPTE time code, etc.
- The latest in graphics programming, featuring algorithms and techniques for texture mapping, hidden-line removal and more.
- TNT (tips, news and tools), a column covering commercial software, books and talk on the networks.
- Programming utilities from PD disks, bulletin board systems and networks.
- Wise buys in new products—from language system upgrades to accelerator boards to editing systems and more.

The fact is, there's no other publication like *The AmigaWorld Tech Journal* available. It's all the tips and techniques you need. All in one single source. So subscribe now and get the most out of your Amiga programming. Get six fact-filled issues. And six jam-packed disks.
Call 1-800-343-0728 or complete and return the savings form below—today!

To order, use this handy savings form.

☐ **Yes!** Enter my one-year (6 issues, plus 6 invaluable disks) Subscription to *The AmigaWorld Tech Journal* for just $59.95. That's a special saving of $35.75 off the single-copy price. If at any time I'm not satisfied with *The AmigaWorld Tech Journal*, I'm entitled to receive a full refund — no questions asked!

Name _____

Address _____

City _____ State _____ Zip _____

☐ Check or money order enclosed. ☐ Charge my:
☐ MasterCard ☐ Visa ☐ Discover ☐ American Express

Account No. _____ Exp. Date _____

Signature _____

Satisfaction Guaranteed!
Or your money back!

Canada and Mexico, $74.95.
Foreign surface, $84.97.
Foreign airmail, $99.95.
Payment required in U.S. funds drawn on U.S. bank.

Complete and mail to:

The AmigaWorld Tech Journal
P.O. Box 802, 80 Elm Street
Peterborough, NH 03458

TJ1191

For faster service, call toll-free 1-800-343-0728.

Clean Up Your Programs

Enforcer and Mungwall help you find your errors before the public ever sees them.

By Carolyn Scheppner

HAVE YOU EXPERIENCED any of the following problems with your own or another company's software?

- The program runs well on your system, but other users report it has problems on their systems.
- The program runs well by itself but has problems running with or after other programs.
- The program runs well most of the time but occasionally crashes or fails for no apparent reason.

Thanks to two powerful new debugging tools—Enforcer and Mungwall—you should be encountering such problems less. When used correctly during product development and testing, these tools (found in the Scheppner drawer) catch the most common causes of these problems—the use of NULL pointers, uninitialized pointers, improperly initialized structures, improper memory usage, and overwritten memory allocations. In fact, many companies now require that all of their in-house software pass Enforcer and Mungwall testing, and have also added this requirement to their contracts for outside development.

HOW THEY CAN HELP

Written by Bryce Nesbitt, Enforcer is an MMU-based debugging tool. An MMU is a memory management unit that can be configured to trap accesses to specified ranges of memory. The 68030 chip has a built-in MMU, and most 68020 boards contain separate MMUs. Because it is MMU-based, Enforcer can trap reads and writes of low memory and nonexistent memory the instant these accesses (also known as "Enforcer hits") occur. This allows you to catch usage of NULL pointers and some uninitialized pointers, and even accesses that would have trashed low memory or otherwise crashed the system. Some of these accesses (such as reads of address 0) may seem harmless on your system, but they could cause your program to fail in the field. If you are developing commercial software (or any software that you plan to distribute), it is extremely important that you invest in an MMU or, at the very least, make sure that your software is tested on machines with MMUs, Enforcer, and Mungwall. As more of the development community begins running these tools, software that is unusable in their presence will be abandoned.

Enforcer is even more powerful when used in combination with Mungwall, a combination memory munging tool by Ewout Walraven that is based on Bryce Nesbitt's Memmung and Randell Jesup's Memwall. The "mung" part of Mungwall fills all of free memory (and all subsequently freed memory) with nasty, odd 32-bit values, such as $DEADBEEF. These values are almost guaranteed to cause serious problems for any program that uses uninitialized pointers or structures, or uses memory or allocations after they are freed. Such usages can occur, for example, when allocations are not freed in the correct order.

Mungwall uses specific nasty 32-bit values in its memory munging to help you diagnose any problems:

- Except when Enforcer is running, location 0 is set to $C0DEDBAD so that programs referencing location 0 will not find a value. Programs referencing location 0 as a string will get a string of high ASCII characters rather than a NULL string, and programs using NULL structure pointers should be irritated into crashing. When Enforcer is running, this is not necessary because, with location 0 containing 0, Enforcer can trap these low-memory accesses by itself.

- On startup, all free memory is munged with $ABADCAFE. If this number shows up, someone is referencing memory in the free list.

- Except when MEMF_CLEAR is set, memory is premunged on allocation with $DEADFOOD. When this appears in an Enforcer report, the caller is allocating memory and doesn't initialize it before using it.

- Memory is filled with $DEADBEEF when it is deallocated, encouraging programs reusing freed memory to crash.

The "wall" part of Mungwall allocates extra memory before and after every memory allocation and fills this "wall" with a fill pattern and other information. On each deallocation, Mungwall checks to make sure that the deallocation size matches the size of the allocation and that the walls have not been overwritten. Mungwall also watches for 0-size allocations, 0-size deallocations, and 0-address deallocations. In addition, Mungwall has an option to "snoop" and report on all memory allocations and deallocations for all tasks or specified tasks. This can be useful when tracking down memory losses. You can then run the voluminous snoop output through the snoopstrip program, which will throw away all matching alloc/dealloc pairs.

Mungwall may be used without Enforcer and on nonMMU machines. If you don't have an MMU, at least test with Mungwall alone. If you are using uninitialized memory or memory after it is freed, Mungwall should help your program to crash immediately (as it might crash on a user's machine

"The best debugging set-up is to connect your Amiga via a null-modem serial cable to another computer running a terminal package with ASCII-capture capability."

when he runs other programs at the same time as yours).

DEBUGGING ARRANGEMENTS

Enforcer and Mungwall both output their debugging information to the serial port at the current baud-rate setting of your machine's serial hardware. After powerup, your serial hardware is set to 9600 baud, but you can modify this by bringing up a terminal package and setting a baud rate. The best debugging setup is to connect your Amiga via a null-modem serial cable to an Amiga or other computer running a terminal package with ASCII-capture capability. Both Enforcer and Mungwall include CTRL-Gs in their output to generate a beep with most terminal packages, and the ASCII-capture capability will allow you to capture all serial debugging output to a file for examination. This is especially useful when combined with serial kprintf() (debug.lib) debugging statements in your code, such as:

```
kprintf("About to close window $%lx\n",win).
```

A clean way to add conditional debugging statements to a C program is to use a MACRO such as D(bug)) by including lines similar to those below in the program. Set MYDEBUG to 1 to turn on debugging. Set bug to printf for printf debugging, to kprintf (and link with debug.lib) for serial debugging, or to dprintf (and link with ddebug.lib) for parallel debugging. The D(bug()) macro is neater in your code because it can be indented and you need not surround it with any #ifdef directives yourself. Just be careful, and remember to put two close parentheses before the semicolon at the end of each D(bug()) statement.

```
/********** debug macros  **********/
#define MYDEBUG 1
void kprintf(UBYTE *fmt,...);
void dprintf(UBYTE *fmt,...);
#define DEBTIME 0
#define bug printf
#if MYDEBUG
#define D(x) (x); if(DEBTIME>0) Delay(DEBTIME);
#else
#define D(x) ;
#endif /* MYDEBUG */
/********** end of debug macros **********/
```

You could then use the macro as follows:

```
win = OpenWindow(&mynewwin);
D(bug("Opened window at $%lx\n", win));
```

If you have only one machine, you can debug to a serial or parallel printer (with Enforcer.par and Mungwall.par). In a pinch, if you have a modem attached to your machine you may have some success doing serial debugging to yourself. If you bring up a terminal package and set it to your modem's baud rate, your terminal program can capture serial debugging output. However, you may lose bytes, especially at low baud rates, if the debugging output is large.

By using Enforcer and Mungwall while you are developing your software, you can catch problems as soon as they are introduced and greatly cut down your debugging time. It is especially useful to place conditional remote debugging statements in your code as you write each routine, so you can quickly turn them on when a problem occurs. You will be able to pinpoint the troublespot easily when the kprintf (or dprintf) output is intermixed with the Enforcer or Mungwall output. The remote debugging commands kprintf and dprintf are available in the linker libraries debug.lib (serial) and ddebug.lib (parallel), respectively. These linker libs are supplied with some compilers, on Commodore's Native Developer Update disks, and in the Scheppner drawer of the accompanying disk. If you prefer, you can also use a source-level or single-stepping debugger in combination with Enforcer and Mungwall when tracking down a problem, to single-step through your code until the hit occurs.

A different low-level method of locating Enforcer hits is to disassemble program memory where the hit occurred (or, if the hit occurred in ROM, to try the nonROM addresses shown in the Enforcer stack dump line), then to match up this disassembly with your own code. When working in assembly, you can just compare the disassembly to your source. Otherwise, you can take the hex values of a sequence of position-independent 68000 instructions near the hit (i.e., no addresses except for offsets and branches) and search for this binary pattern in your object modules. If you find the pattern, do a mixed source and object disassembly of that object module (for example, with SAS' OMD you could OMD >ram:dump mymodule.o mymodule.c) and then look in the output for instructions matching those where the hit occurred. ►

Figure 1: Sample Enforcer Output

```
Program Counter (approx.)     = 343F4A     Fault address          =     14
User stack pointer            = 348734     DOS process address    = 339590
Data:  DDDD0000  DDDD1100  DDDD2200  DDDD3300  DDDD4400  DDDD5500  DDDD6600  DDDD7700
Addr:  AAAA0000  AAAA1100  AAAA2200  AAAA3300  AAAA4400  AAAA5500  AAAA6600  00002E28
Stck:  00210D70  00000FA0  00339F84  BBBBBBBB  BBBBBBBB  BBBBBBBB  BBBBBBBB  BBBBBBBB
READ-WORD (- - -)(-)(-)  SR=0008  SSW=0161
Background CLI, "lawbreaker", Hunk #0, Offset $5A
```

ENFORCER HIT EXPLANATIONS

Enforcer gives you lots of valuable information to help debug your program's hits. Consider the sample Enforcer hit caused by a program called lawbreaker shown in Figure 1. Here is an explanation of the most important items:

Program Counter: The memory address at which the program was executing instructions when the hit occurred. For some types of hits this often will be the address of the instruction after the hit. Note that if your program passes a bad pointer or an improperly initialized structure to a system ROM routine, you may cause the ROM code to read or write to an illegal address.

Fault Address: The address where the illegal access occurred. In this example, the illegal access occurred at address $14, and, as specified later in the debugging output, this access was a READ-WORD access. Therefore, the illegal memory access was an attempt to read a WORD (two bytes) at address $14. Low-memory accesses are often caused by NULL pointers to structures. If, for example, your code or a ROM routine references a structure member at offset $20 and you provide or use a NULL structure pointer, Enforcer will pick up a hit at address $20.

Register Dump: Shows the contents of the program's registers and stack at the time of the hit. This information can help assembly programmers and programmers who like to debug at a low level.

Access Type: In this example, the access was a READ-WORD and probably accessed a WORD-sized structure member. A READ-BYTE access is generally caused by a bad string pointer, while a READ-LONG is usually caused by a bad pointer or a bad pointer within a structure. WRITE-WORD, WRITE-BYTE, and WRITE-LONG accesses indicate that you are causing memory to be trashed and can be caused by bad pointers or bad code. Occasionally you will see an INSTRUCTION access of illegal memory, generally caused by trashed code, a trashed return address on your stack, or an invalid library base.

Program Name and Hunk Offset: The program name is the name of the task or command that was executing when the hit occurred. If possible, Enforcer also provides a hunk offset to the program counter's reading if the hit occurred within the program's own code instructions.

SAMPLE MUNGWALL OUTPUT

Mungwall provides a similar volume of debugging details. Study the hits by a program called mungwalltest that are shown below. Following each hit I added an explanation in parentheses. For reference, the arguments for memory functions are AllocMem(size,type) and FreeMem(address,size). The A: and C: addresses are Mungwall's guesses at the address from which AllocMem() was called. A: is the address for an assembler caller. C: is the address for a C caller, assuming a standard stub. Because Mungwall is wedged into the memory allocation functions, it can only guess the caller's address based on what is pushed on the stack. The "at" address on the first line of a Mungwall hit is the task address of the caller. Note that Mungwall has special code to prevent trapping the partial (wrong size) deallocations that are performed by layers.library. If any other debugging tools are also wedged into AllocMem() and FreeMem(), Mungwall's A: and C: addresses may be thrown off by additional information pushed on the stack, and Mungwall will also be unable to screen out partial layers deallocations (which will show up as hits on your task's context).

```
AllocMem(0x0,10000) attempted by `mungwalltest' (at 0x339590)
   from A:0x35C03A C:0x35677E SP:0x35CFC0
(tried to allocate 0 bytes of memory)

FreeMem(0x0,16) attempted by `mungwalltest' (at 0x339590)
   from A:0x35C068 C:0x3567C4 SP:0x35CFB8
(tried to free memory with a NULL pointer)

FreeMem(0x33BD10,0) attempted by `mungwalltest' (at 0x339590)
   from A:0x35C068 C:0x3567D4 SP:0x35CFB0
(tried to free 0 bytes of memory)

Mis-aligned FreeMem(0x33BD14,16) attempted by `mungwalltest' (at
0x339590)
   from A:0x35C068 C:0x3567E2 SP:0x35CFA8
(deallocation address known incorrect because not aligned like alloc)

Mismatched FreeMem size 14!
Original allocation: 16 bytes from A:0x35C03A C:0x3567A0 Task
0x339590
Testing with original size.
(deallocation size does not match allocation size)

19 byte(s) before allocation at 0x33BD10, size 16 were hit!
>$: BBBBBBBB BBBBBBBB BB536572 6765616E 74277320 50657070
65722000
(program trashed bytes that precede its allocation)

8 byte(s) after allocation at 0x33BD10, size 16 were hit!
>$: 75622042 616E6400 BBBBBBBB BBBBBBBB BBBBBBBB
BBBBBBBB BBBBBBBB
(program trashed bytes that follow its allocation)
```

As you can see, Mungwall alone can catch a large variety of memory-related software problems. But one of the most important benefits of Mungwall is that by filling freed memory with nasty 32-bit values, it can force subtle memory mis-

Continued on p. 63

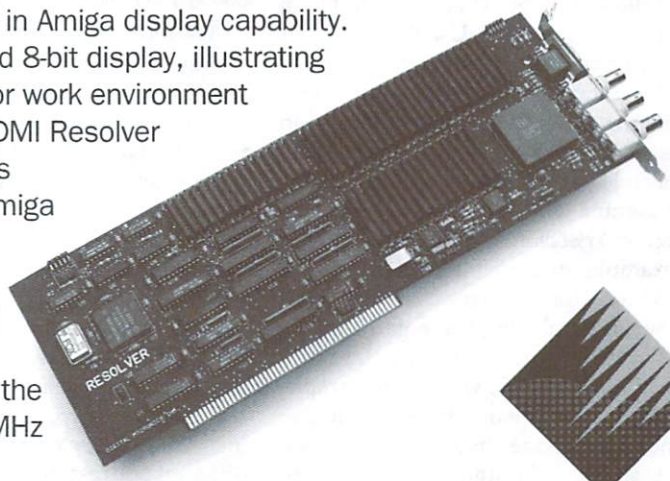
Don't be Fooled by any other Solution. 1280x1024 Resolution.

DMI Resolver™

- 1280x1024 Resolution
- 8-bit Color Graphics
- 16-million Color Palette
- 60MHz Processor
- Programmable Resolution

The DMI Resolver™ graphics co-processor board offers a new dimension in Amiga display capability. Shown above is an unretouched 8-bit display, illustrating the 1280x1024 resolution color work environment provided by the Resolver. The DMI Resolver boosts the display and graphics processing capabilities of all Amiga A2000 and A3000 series computers, under both AmigaDOS and UNIX operating systems. Not to be confused with a frame buffer or grabber, the Resolver is a lightning fast 60MHz graphics co-processor.

Whatever your application – desktop publishing, presentation graphics, animation, 3D modeling, ray tracing, rendering, CAD – let the Resolver move you into a new realm of resolution and workstation quality display.



Digital Micronics, Inc.
5674 El Camino Real, Suite P
Carlsbad, CA 92008
Tel: (619) 431-8301 • FAX: (619) 931-8516
Call for more information and the dealer nearest you.

Circle 6 on Reader Service card.

Efficient Assembly Programming

By Jamie Purdon

WHEN DESIGNING AND and writing "commercial" code, the money is in getting the job done. This means developing good, high-level design, coding plans, and sticking with a schedule (even when it's self-designed and imposed). The fun part is writing the low-level, hardware-banging code.

You can program in a system-friendly manner, yet still "bang on the hardware." One way is to keep the Blitter reserved with OwnBlit() and DisownBlit() for only extremely short periods of time. Another example is to limit your mouse handling (mousebutton and mousemove events) to IDCMP messages.

Still, sometimes you need to do something nasty. For example, to plot brushstrokes much faster than Intuition will give mouse-position reports, you need to read the mouse hardware. In this case, be sure to take into account Intuition's ActiveWindow (be sure it's one of yours) and make sure that you have replied to all outstanding IDCMP messages. In other words, make sure that the system has not queued up any input (messages) before you read from an input device.

The general idea is to use high-level ROM routines and to write the fastest-executing assembly language routines. Using the built-in libraries makes efficient use of a resource that's always present. It allows for very small code and high functionality. The ROM routines are free. Use them.

Assembly language programming can be as efficient as that of most high-level languages. You can code good applications, that execute somewhat slowly, by using the built-in routines whenever possible. Of course, you gain the most speed (at the expense of code size) when you "put the pedal to the metal" and use assembly language.

Use assembly language for speed or not-provided-for (low-level) capabilities. For example, an application might contain a screenful of gadgets with custom imagery. Now, Intuition seems to take forever when displaying these gadgets, because so many separate blits occur. Essentially, Intuition makes repeated calls to DrawImage(), which uses graphics.library, which uses the Blitter. A workable solution is to use the complement mode for gadget highlighting and no imagery for the gadget, as far as Intuition cares. You can write custom code to draw the gadget imagery all at once. These routines will usually run quicker than Intuition—unless, of course, you use repeated calls to DrawImage(). You still get the high-level advantage of letting Intuition handle the gadget interaction, but add the low-level advantage of decreased gadget-rendering time.

COUNTING CYCLES

Cycle counting is an optimization technique that lets you eke the best performance out of your Amiga's CPU. Often tedious, it involves looking up (or memorizing) the timing, in cycles, of every instruction. Cycles are the number of "ticks" of the system clock that each instruction takes. They are always in multiples of two; the minimum cycle count is four. (Even a NOP:No Operation instruction takes four cycles to execute.) I'll go into more detail later, but I want to emphasize now that a stock Amiga can execute a theoretical maximum of only 1,750,000 (7MHz/4) instructions per second. (Four cycles is the shortest cycle count.) Real-world timing averages are often much worse than this. When working with graphics, it is ideal to update the screen 60 times per second, however, there are usually not enough cycles in $^1/_{60}$ of a second.

Many people never get around to cycle counting when programming the Amiga. Besides that drudgery, the dearth of programming utilities that display individual instruction cycle timings makes it difficult. You usually end up consulting a paper reference. I use the *MC68000 16-/32-bit Microprocessor Programming Reference Card* published by Motorola. It has all the instructions on one side of a $8^1/_2$×25-inch fold-out card.

68000 TIMINGS

Assembly language is a natural for cycle counters because what you code is what you get. A 680×0 instruction divides into three parts: the *opcode*, the *source address*, and the *destination address*. Either (or both) of the addresses can be CPU registers. Each instruction has many variations that differ in the operands used. Programmers control the operand selection, and smart ones consider this choice when optimizing. The operands used directly affects how long it takes an instruction to execute.

There are some tricks to remembering cycle timings. Some instructions operate on only one address (or register) and have only two parts, an opcode and a combined source and destination register. Each part of an instruction takes a specific number of cycles (ignoring cache considerations). Most instructions have many variations when you consider all the different addressing mode possibilities. It's easiest to develop some general rules for (address and) memory-access timing. Each WORD-size (16-bit) memory access "costs" two cycles. LONG (32-bit) memory references cost four cycles (a LONG is two WORDs). It's noteworthy that these timings are based on memory access considerations (and are reduced when a cache is available).

For the best memory-access timings, I recommend using registers because, they do not cost any extra cycles for WORD-size operations. Many LONG-size (register-to-regis-

Fast and productive techniques of coding for the various Amiga CPUs and dealing with system libraries.

ter) operations, however, cost an additional four cycles. For example, ADD.W d0,d1 takes four cycles, while ADD.L d0,d1 requires eight cycles. Essentially, the CPU is a 16-bit machine. It requires extra time to deal with LONGWORDs. The 68000 is called a 16-/32-bit CPU because it is most efficient with 16-bit data, but the memory addressing architecture is set up for 32-bit operations.

Don't take these numbers for granted. Occasionally review the cycle counts of all instructions. This will help you plan code sequences with minimal cycle counts. You'll find that register-to-register MOVE instructions take the same number of cycles for both WORD and LONG operations.

For efficiency, the Scc instructions come in handy when dealing with flag bytes. They act very much like the Bcc instructions. Instead of branching, however, they set or clear a byte. This byte can be in memory or (the lowest byte in) a Dn data register.

Another common trick is to use the MOVEM instructions wherever possible. Note that a MOVEM.W into an An address register will sign-extend the top WORD. Another way to think of this is that a MOVEM.W will destroy the upper WORD in an A register.

After you've written some code and are optimizing it, look for substitute opcodes (with quicker timing and fewer cycle counts). An obvious substitution is replacing ADD # instructions with ADDQ #. Another is to use:

MOVEQ #0,Dn

(four cycles) in place of:

CLR.L Dn

(six cycles). There are more subtle tricks you can use: ROXL.L (ten cycles) is often replaced by ADDX.L (eight cycles).

SUBA An,An

(eight cycles) is a common instruction for clearing an address register. If you have already cleared a data register, however, you can replace the instruction with:

MOVEA.L Dn,An

(four cycles). If you would rather optimize for small code versus execution time (the classic trade-off), then a good instruction to use is:

MOVEM.L Zeros,Dn-n/An-n

All this requires is that Zeros be defined as an array of n (equaling the maximum number of registers) zero-value LONGWORDs. It's a cycle hog because it costs eight cycles to clear each register (plus 12 cycles of instruction overhead). But this opcode requires only four or six bytes of code space—depending on the Zeros' addressing mode—for any number of registers to clear.

ACCELERATED AMIGAS HAVE CACHES

A beneficial habit is to design code and data structures that function very efficiently when executed on cache-bearing CPUs, but to use only opcodes that work on all CPUs (68000 instructions). To take advantage of newer CPUs without providing separate code for each CPU, you must take into account caches when unrolling loops and designing data structures. This provides good performance on a 68000 and better than expected improvements (if you consider only clock-speed differences) on newer CPUs.

A cache is a piece of hardware that sits between the CPU and the memory. It acts as a very fast (but small) memory and a lookup table. The lookup table keeps track of the last n memory accesses. Cache memory has the benefit of allowing much quicker access to memory than is possible with regular memory subsystems. Frequent accesses to the same memory location are remembered by the cache and the data is available to the CPU much quicker than if an actual memory card access occurred. This really helps on Amiga 3000s that are upgraded with older, 16-bit memory cards.

The 68020 has a code cache, while the 68030 has both code and data caches. Note that register-based algorithms are still fastest (for data manipulation). The 68030's data cache will not improve performance for these instructions. The 68020's code cache is 256 bytes arranged as 64 LONGWORDs. The 68030's two caches are each 256 bytes long as well, but are organized differently. Each holds 16 groups of four LONGWORDs.

DATA STRUCTURES

To take advantage of the 68020 and 68030, group often-used variables within the same mod 16 address range. The 68020 deals with memory in units of four bytes (one LONGWORD). With the 68030's (very fast) burst mode enabled, data and instructions are fetched in groups of 16 bytes (four LONGWORDs). The cache is loaded with all four LONGWORDs in the mod 16 grouping. (The hardware is optimized to fetch the desired LONGWORD first.) Grouping variables (and code sections) based on mod 16 grouping allows downward compatibility while taking advantage of the 68030. If you're optimizing strictly for the 68020, disregard data-cache considerations and stay with LONGWORD (as opposed to WORD) alignment for the code sections: The '020 fetches in ►

groups of LONGWORDs (the '030 default), but has no burst-mode facilities like the '030.

Remember that, although 68020 upgrades are fairly inexpensive (compared to 68030s and 68040s), the Amiga 3000 comes with a 68030. You'll probably want to optimize for the 68030 at some point. (It also comes with a math chip, but I digress...)

Keep in mind that the '030 data cache does not work with chip RAM. Chip RAM addresses are never cached, because the cache can contain invalid data if any of the custom chips happen to change chip data.

UNROLLING LOOPS

Unrolled loops are much more readable when coded with macros. You specify a macro to represent the innermost loop code, then type the macro as many times as the loop is to execute. For example, if you want a loop to execute four times, you might code:

```
   moveq #4-1,d0  ;loop counter, '-1' for dbxx loop
loop:
   INNERLOOPCODE  ;macro
   dbf d0,loop
```

This same loop, unrolled, would look like:

```
INNERLOOPCODE  ;macro
INNERLOOPCODE  ;macro
INNERLOOPCODE  ;macro
INNERLOOPCODE  ;macro
```

At the expense of extra code space, this executes much more quickly, because the dbf instruction is never executed. You must decide how you feel about the code-size versus execution-time trade-off.

Loops that iterate for a multiple of two are easiest to unroll. Many loops are not so easily unrolled. The loop counter is divided by the number of inner-loop iterations that are unrolled. For example, if the above loop were to execute 320 times, you could code it as:

```
   move.W #(320/4)-1,d0  ;loop counter, /4 ( inner loop expanded)
loop:
   INNERLOOPCODE  ;macro
   INNERLOOPCODE  ;macro
   INNERLOOPCODE  ;macro
   INNERLOOPCODE  ;macro
   dbf d0,loop
```

This is still more efficient than an unrolled loop, because the dbf instruction executes only $^1/_4$ as often.

A good rule to follow is: Unroll loops only until the code size approaches 128 bytes. This allows repeatedly called subroutines to remain in the cache. Also, the calling code will most likely remain in the cache. Another reason to keep unrolled loops small is that a BRA instruction at the bottom of a loop can be the short form (BRA.S), which executes quicker. This short form requires a (signed) seven-bit (128-byte) offset.

CPU AND MATH-CHIP CONSIDERATIONS

Every CPU in the 680x0 family recognizes the basic 68000 instruction set. Each newer CPU has new instructions that will not work on the earlier CPUs. I recommend that you ignore these instructions (unless you wish to write CPU-specific code) until a very low-cost 680x0 machine is available. I don't recommend providing code that is optimized for a 68030 (for example) and somewhat crippled on a 68000, unless you wish to spend the time and effort to write and test code for different CPUs.

Commodore's math libraries let you write just one piece of code that will run on all math chips. However, they have the penalty of subroutine overhead. They also cost extra programming and testing time. You can eliminate much of the overhead by using the CPU's integer math instructions. Except for certain high-range applications, such as CAD or ray tracing, CPU-based math is usually faster than that on a hardware math accelerator. I recommend using nonCPU-specific integer math whenever possible.

WHAT ABOUT FLOATING-POINT MATH?

I often use a combined fixed- and floating-point method to keep track of the number of binary fractional bits. Binary fractions are not so hard to deal with: They work according to the same rules as decimal fractions. You can add and subtract only numbers with the same number of fractional bits. You can multiply any two numbers, however, as long as you remember that the result contains n fractional bits—where n is the total of the number of fractional bits of both operands. You can add and remove fractional bits with one (shift-type) machine language instruction.

Arguments in favor of (CPU) integer-based math hold true for the 68040. On board the CPU, the 68040 has a math chip that allows floating-point math functions to execute faster (in fewer cycles) than any outboard math chip. The integer functions are still faster. If you can get away with 16- or 32-bit precision, it's still faster to use integer (680x0-generic) functions. Plus, this offers the benefits of being CPU-compatible and following my philosophy of coding for a 68000 but making use of 680x0 CPU's "transparent" features. Also, in assembly language, integer math is a natural, it is common to the whole 680x0 family.

One caveat with fixed- and floating-point fractions is to not spend a lot of cycles removing fraction bits. Beware that the shift instructions are relative cycle hogs: They typically take 6+2*number_of_bits cycles. A good way to avoid this overhead is to remove bits (by shifting) only infrequently. Another trick is to use the four-cycle add instructions for left shifts.

The fixed- and floating-point method works very well with image processing applications. Most image-processing algorithms deal with eight-bit source operands: RGB colors are normally stored in eight bits per component, and blending, shading, and antialiasing algorithms usually deal with eight or fewer bits for the "blend factor." Even when you add up the bits, two eight-bit sources only (multiply to) yield a 16-bit number—which conveniently fits in a WORD sized result and is optimally a CPU register. Here's an example that blends two numbers in a 60/40 ratio:

```
nbits  set 8
percent  set 60
altblend set (1<<nbits)*(100-percent)/100
mainblend set (1<<nbits)-altblend


   mulu #mainblend,d0  ;A blend, result has fraction
   mulu #alt,d1    ;B blend
   add.W  d1,d0    ;(A*frac)+(B*(1-frac)) = result <<nbits
   asr.W  #nbits,d0  ;remove nbits = 60%d0 + 40%d1 .byte valid
```

How well does all this theory convert real-world examples? Below (and in the Purdon drawer of the accompanying disk) are some examples of routines I use over and over in my projects. Let's examine how they work.

The "CUSTOMIZE" comments refer to specific lines of code that are application dependent. You can use absolute- or address-register-relative (stack-pointer or base-page) addressing, but I recommend a base-page setup. Base-page addressing allows for WORD-size (16-bit) addressing, which is quicker than absolute addressing because, in terms of memory cycles, only a WORD (not a LONGWORD) needs to be accessed when the CPU fetches an instruction.

CHECK-FOR-MESSAGE MACRO

A macro, the CHECK4MSG routine checks for a pending message. I use it within sections of code that I may want to abort without spending much time checking for an abort condition. The standard method is to simply check your message port's signal bit, but this tells you only if messages are pending. Often you may not want to abort unless a specific message is detected. Standard methods do not tell how many messages are pending, nor provide any useful information about the types of pending messages. CHECK4MSG works differently: It checks for a message, removes it from the message port, and provides the ability to scan the incoming message list. Of all the built-in routines CHECK4MSG is closest to Exec's GetMsg() library call. Functionally it is slightly different, but it offers the obvious advantage of quicker execution, because it's a macro, not a subroutine call.

Remember, subroutine calls generally take 34 cycles. The breakdown is: An RTS instruction takes 16 cycles, and a typical call-subroutine instruction takes 18. (BSR takes 18 cycles, as does JSR _LVOwhatever(a6). JSR abs.l requires 20 cycles.) So, plan on spending at least 34 cycles every time you call a subroutine. The following code is so short that, if it were a subroutine, the 34-cycle overhead would represent a significant portion of the execution time. Take a look:

```
CHECK4MSG: MACRO
   lea OnlyPort_(BP),A0  ;a0=msg port adr, CUSTOMIZE
   lea MP_MSGLIST(A0),A0  ;TOP of list
   cmp.l  8(A0),A0   ;beq if empty
   ENDM
;
;example usage
;  CHECK4MSG
;  bne abort_have_a_msg
```

While this is really a textbook example, I want to point out the options that it makes available. One of the neatest is the ability to prescan the message list, letting you scan for a cancel code without removing any messages from the list (very user-friendly). It also enables you to control the time at which a message is removed from the pending-message list, because you do the removing. You can remove one without affecting any of the other messages, which is handy when you want to get rid of a certain class of messages—such as mousemoves—without affecting rawkey or mouse-button events.

If you implement prescanning of a message list, you will probably want to use Intuition's rawkey messages. Of course, you should also use console.device's RawKeyConvert() routine to decipher the keypresses, making your program compatible with the user's Preferences keymap. There are times, however, when you may simply want to check for a keycode without calling RawKeyConvert(). Certain keys—the function keys, the spacebar, and the arrow keys—seem to retain the same codes, with all keymaps, on most Amiga models.

If you do program for these specific rawkey codes, be warned that your application will not be CDTV friendly. CDTV lacks a keyboard, and the CDTV remote control's arrow keys function as move-the-mouse keys. They don't necessarily return "nice and programmer-friendly" arrow (raw) key events.

An acceptable method is checking your message list ahead of time (prescanning) and removing messages from a message port before they are called by GetMsg(). This is based on the idea that once a message is processed with PutMsg() it is the property of the task that owns the message port. Plus, it has the added advantage of removing the drudgery of maintaining a list of incoming events from the application. Somehow this has to be performed if you want to respond to each incoming message in the order that the user generated them. (The messages might be a list of many menu-equivalents or gadget events.) Exec does this for you, automatically.

Because you own the message list on your ports, you may delete individual messages without actually calling GetMsg(); simply use Exec's REMOVE macro. REMOVE can substitute for GetMsg(), but you must be sure to still use ReplyMsg() for each one.

Another nice thing about messages and using only one port for all of them (a design trick) is that you maintain synchronicity—you can respond to events in the same order as they occur. Signals have no such capability. Signals tell you only if something happened since you last cleared the signal. If multiple signals occur, you have no way of knowing the order in which they occurred. A signal is simply a bit in your task structure. It will not tell you if something has happened more than once. It will only tell you if the fact that something has happened. When using message port signals, you never know if more than one message has been received.

FREEONEREMEMBER ROUTINE

The Exec and Intuition ROM libraries provide two memory-allocation schemes. The memory-allocation calls are very similar, but Exec's FreeMem() and Intuition's FreeRemember() are different: FreeMem() frees one chunk of memory, while FreeRemember() frees potentially many chunks of memory. While Exec does provide routines that deal with memory management in a more sophisticated way (the Alloc/FreeEntry() routines, and so on), Intuition's routines are more elegant. The standard Intuition library calls, however, do not provide for selectively freeing just one chunk of memory. You can invoke FreeRemember() only for all the memory-allocation chunks.

In ordinary programming, Intuition's Alloc/FreeRemember() routines are rare, because they provide no way to deallocate one specific chunk of memory. The code below, FreeOneRemember(), is a routine that frees just one memory allocation from an Intuition Remember list. It is a high-level routine; cycle-counting is ignored and utility and code compactness are emphasized. The 680x0 CPU family is very adept at list handling.

Essentially, FreeOneRemember():

- Finds the memory in a Intuition Remember list. ►

- Removes the Remember chunk from the list.
- Creates a dummy Remember structure/pointer.
- Does a FreeRemember() of the dummy structure.
- Deletes the dummy structure.

This routine retains the FreeRemember() advantage of being able to free all memory allocations at once, which is handy when your application quits or aborts. With this capability, you do not have to call many separate "free-some-memory" routines, and your code runs faster. The routine provides a new advantage in that you can deallocate just one chunk of memory, and still not have anything more than the standard Remember structure overhead. Also, when using FreeOneRemember(), you need not keep track of the size of the memory chunk, which Exec's FreeMem() requires as an argument. You simply pass it one argument—the address of the memory.

If you want to maintain separate Remember lists, you can customize FreeOneRemember() to require two arguments: a memory address and a Remember list address.

FreeOneVariable() is a slightly higher-level routine: You pass it the address of a LONGWORD variable that contains a pointer to the memory. A NULL pointer is fine. It's a painless way to ensure that a piece of memory is deallocated.

```
FreeOneVariable:  ;A0=Address of variable to free, RETURNS a0
  ;unmolested
    move.l (a0),d0 ;address of memory to free
    clr.l  (a0)  ;(say it's gone...)

FreeOneRemember:  ;D0=Address of memory to free,
    tst.l  d0     ;address to free mem?
    beq.s  finalend_f1r   ;none...get outta here
    movem.l d0/a0/a1/a6,-(sp)  ;DESTROYS D1

    move.l #RememberKey,a0 ;address of Remember list (CUSTOMIZE)

f1restart: ;TOP OF MAIN/SCANNING LOOP
    move.l a0,a1      ;a1=save prev for de-linking
    move.l (a0),d1    ;d1=rm_NextRemember
    beq.s  endof_f1r  ;nothing in list (why?...)
    move.l d1,a0      ;a0=next/current
    cmp.l  rm_Memory(a0),d0  ;this chunk aptr to our memory?
    bne.s  f1restart  ;nope...reloop till endalist

       ;FOUND IT, FREE THE CHUNK
f1gotm:  ;REMOVE FROM REMEMBER LIST
    move.l (a0),(a1) ;prev<==NEXT after me ;de-link me...ao=me, a1=prev
    clr.l  (a0)   ;rm_NextRemember(a0) ;points to nOOne, now.
       ;FREE THIS 'REMEMBER' STRUCT & ITS MEMORY CHUNK
       ;USE/BUILD DUMMY REMEMBER KEY/LIST
    move.l a0,-(sp)  ;temp pointer to a remember struct
    lea (sp),a0   ;'pointer to a remember pointer'...
    CALLIB Intuition,FreeRemember
    lea.l  4(sp),sp  ;de-stack temporary remember struct

endof_f1r:
    movem.l (sp)+,d0/a0/a1/a6
finalend_f1r:
    rts
```

QUICKCOPY ROUTINE

Many graphics programs need an undo buffer. Of course, on the slowest, unaccelerated generic machine, the Blitter is the fastest engine to use when copying a bitmap. When lots of fast RAM and a faster CPU are available, however, there are advantages to copying to an undo buffer that exists in fast RAM. This involves using the CPU, because the Blitter can work only in chip RAM.

The Exec's memory-copy routine is fairly efficient, but faster routines are possible with a little bit of effort. Exec's routine uses the MOVEM trick, but it's in a very small, tight loop. A faster routine results from unrolling the innermost loop only far enough to still fit inside an 020's cache. Macros can help with the unrolling. In this case, a macro is used in the innermost loop. The macro expands to many instructions, but the code remains clear because the start and end of the loop are visible on an editor screen. (With the macro, the loop code is only a few lines long.)

```
QUICKCopy: ;d0=count, a0=from address a1=to adr
    movem.l d0-d7/a0-a6,-(sp)

    cmp.L  #48+1,d0
    bcs.s  qclast

    move.l #384,-(sp)
    bra.s  qcckIp
copyfirstloop:
    copy384
qcckIp:
    sub.L  (sp),d0
    bcc.s  copyfirstloop
    add.L  (sp)+,d0
```

A macro, copy384, copies 384 bytes using MOVEM.L instructions. All registers are filled with copy data except for A0 and A1, which contain the addresses, and D0, which contains the current copy count (–384). Note that the complete code is in the Purdon drawer on disk, the above is a fragment only.

Using the Exec copy routine will work, but my custom routine will outperform it for large copies, which most graphics code seems to need.

QUICKCopy requires at least WORD alignment of both source and destination addresses. It will perform better (on accelerated machines) if LONGWORD alignment is used. Byte-string copies, where alignment is not guaranteed, are best handled by another routine.

To see how these routines can work together, study the demo program in the disk's Purdon drawer. Combining FreeOneRemember(), CHECK4MSG, QUICKCopy, and QuickPlot, the demo plots a bouncing ball on the top half of the screen, and copies the top half to the bottom half after each new pixel is plotted.

After you put in the extra effort to write efficient assembly routines, consider making your routines available for other programmers to use. Package the code into a library, or device or at least make it accessible from ARexx. Your fellow assembly programmers (like me) definitely will thank you for the help. ■

Jamie Purdon is the author of NewTek's DigiPaint and ToasterPaint software. He's made a career of the Amiga, programming in 68000 assembly language for over five years. Contact him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or on BIX (jamiep).*

Become a part of the *AmigaWorld* Programming Team

We're looking for quality programs to support the growth of the *AmigaWorld* product line, and we need your help.

▼

GAMES
ANIMATION
3D
UTILITIES
CLIP ART
AMIGAVISION APPLICATIONS
OTHER STAND-ALONE APPLICATIONS

▲

We offer competitive payment and an opportunity for fame. Send your submissions or contact us for guidelines:

Amiga Product Submissions
Mare-Anne Jarvela
(603) 924-0100
80 Elm Street
Peterborough, NH 03458

Custom Interfaces With ARexx

By Marvin Weinstein

AREXX, AN INTERPRETED language, is not well suited to writing computationally intensive programs. It is, however, perfect for creating interfaces to programs that are not particularly user friendly. Implementing a user-friendly, fully Amiga-style front-end to a program requires access to the graphical user interface (GUI) not provided by the standard ARexx support libraries that come with OS 2.0.

Fortunately, you can obtain a number of ARexx-shared libraries and manipulate the Amiga's GUI from within an ARexx program. Some of these libraries are part of commercially supported packages; others are freely distributable. In a previous article (see "Extending ARexx," p. 18, Oct. '91), I explained how to use the automatic requesters in Willy Langeveld's freely distributable rexxarplib.library to interact with the user.

While automatic requesters are simple to set up and can provide wonderful results, designing a really nice interface requires more flexibility than the requesters provide. If you want to go the extra mile, rexxarplib provides tools for creating an interface from the ground up. This article designs such an interface for the archiving program, Lharc—a perfect example of a useful program that needs a simpler interface.

Using Lharc sporadically, combined with my terrible memory, forces me to continually reread the help information and need several attempts to get the correct preface symbols, in the correct case and in the correct order. My solution had been to limit myself to extracting the contents of an archive and ignoring Lharc's other capabilities, but what I really wanted was an easy-to-use interface to Lharc. With the more advanced features of rexxarplib I created such an interface. (If you like my interface, then you get not only a lesson in ARexx programming, but also a useful utility. If you dislike my interface, then by the end of this article you will have all the tools required to rebuild it to fit your needs.)

DESIGN REQUIREMENTS

I decided that the Lharc interface must be a gadget-laden panel that opens on Workbench. It would have to provide a way to set the archive name by means of a file requester or a string gadget. Similar methods would be employed to define the destination directory for unpacking an archive and to set the search pattern used in the process. There would be a column of buttons to set Lharc's switches, and these would toggle on and off to indicate their current state. In addition, there would have to be a second column of buttons to execute Lharc commands.

ABOUT THE PROGRAM

In my previous article, I presented a program called fastmenu.rexx that creates a window endowed with button gadgets. The program in this issue, rexxlharc.rexx, is a straightforward, but more complicated, extension of the first one. Although rexxlharc.rexx uses more rexxarplib commands, complications arise because this program handles all messages generated by clicking on button gadgets, whereas the gadgets used in fastmenu.rexx communicated directly with the REXX host process.

This article will touch on the most important aspects of rexxlharc.rexx. It is meant to be read in conjunction with the listing in the accompanying disk's Weinstein drawer. The drawer also contains a copy of rexxarplib.doc, a complete, albeit telegraphic, description of the syntax of all the commands provided by rexxarplib.library. With the explanations given here and the comments included in the listing, the parts of rexxarplib.doc having to do with opening windows and endowing them with gadgets should be accessible to you.

INSTALLING THE PROGRAM

To run this program, first copy the file rexxlharc.rexx into your rexx: directory. If you lack current versions of rexxarplib.library and arp.library, copy them from the disk into your libs: directory. Finally, copy Lharc to your C: directory or elsewhere in your search path. Once this is done, you can type:

```
rx rexx:rexxlharc
```

If you do not wish to tie up a CLI, type:

```
run rx rexx:rexxlharc
```

Note that if you are running WShell, there is no need for the rx command.

WHAT THE GADGETS DO

Once the program is running, it can be shut down by clicking on the close gadget in the upper-left corner of the window. The gadget labeled Archive Name: causes a file requester to open. If you want to create a new archive, use this requester to define both its path and filename. You can omit the .lzh at the end of the archive's name—the program will append it. You can, of course, avoid the file requester and type the full name of the archive directly into the adjacent string gadget.

To select the directory where files will be placed when unpacking an archive, click on the gadget labeled Destination:. This will open another file requester. As with the Archive Name: gadget, you can ignore it and type the name of the destination directory directly into the adjacent string gadget.

With the rexxarblib.library, and a little ingenuity, you can add a GUI to Lharc or any other program.

(Note that the form of this requester will differ under AmigaDOS 1.3, where rexxarplib uses the ARP file requester, and 2.0, where it uses the ASL file requester. See the listing for comments.)

The gadgets that appear in the column labeled "Switches" are used to set Lharc options; highlighting indicates options currently in effect. Some of Lharc's switches require arguments. Clicking on the associated gadget brings up an automatic requester that lets you supply the required information. Once the desired switches have been set, you can click on a command to launch an instance of Lharc. Each click on a command gadget causes a new CLI window to open. All CLIs are fully interactive, so you can abort any instance of Lharc by activating the appropriate window and typing CTRL-C.

HANDLING THE MESSAGE PORT

Let's consider a skeletal form of rexxlharc.rexx to see how the program handles rexxarplib host-generated messages:

```
/** rexx:rexxlharc.rexx - Version 1.0 * **/
   call addlib("rexxarplib.library",0,-30,0)
      .
      .
      .
/*
* Before doing anything open your message
* port. If this fails exit cleanly
*/
testport = openport(LHARCPORT)
      .
      .
      .
/*
* Asynchronously launch a string ...
*/
address AREXX ,
" 'call createhost(LHARCHOST,LHARCPORT)' "
      .
      .
      .
call MakeWindow()
      .
      .
      .
/*
* Everything is ready, ...
*/
quitflag = 0
do forever
   if quitflag = 1 then leave
   t = waitpkt(LHARCPORT)
   /*
   * This loop handles all currently queued
   * messages then goes to sleep again.
   */
   do ff = 1
      p = getpkt(LHARCPORT)
      if c2d(p) = 0 then leave ff
      command = getarg(p)
      t = reply(p,0)
      select
         when command = CLOSEWINDOW then do
            call CloseWindow(LHARCHOST)
            quitflag=1
         end
            .
            .
            .
         when command = LISTARCHIVE then do
            $vdots
         end
            .
            .
            .
         otherwise nop
      end
   end
end
exit
```

SALIENT FEATURES

The program begins by adding both rexxsupport.library and rexxarplib.library to ARexx's internal list (a process discussed in the Oct. '91 article). The first important fragment of code is the call to ARexx's openport() function, which creates a public message port with the name LHARCPORT. Here it will receive messages from its rexxarplib host. The code that follows checks to see if ARexx succeeded in opening the message port; if it failed to open, the program exits.

If all went well, the next major step is to create the rexxarplib host, LHARCHOST. A rexxarplib host is a separate process that knows how to open a window and endow it with menus and gadgets. The host monitors IDCMP messages generated by clicking on gadgets attached to the window. When a message arrives, it creates an ARexx message, ►

fills its slots with the information specified in the call to AddGadget(), and then sends it off to LHARCPORT.

Rexxarplib hosts are created with the createhost() function, which accepts two arguments. The first specifies the name the host should use for its own public message port; the second is the name of the port to which it will forward the ARexx messages it constructs. Enclose the call to this function in two sets of quotes and preface it by the address AREXX instruction (simply inserting it in the program at this point would halt execution). For historical reasons the call to createhost() does not return until the host closes down. To avoid this lockup, send the call as a string program to AREXX (thus automatically running it as an asynchronous process). Two sets of quotes are needed: The REXX interpreter eats the first set in parsing the line, and the second set lets AREXX recognize the string as an in-line ARexx program.

Once LHARCHOST exists, you can tell it to open a window and then adorn it with various gadgets. This is done through the subroutine MakeWindow() (discussed below). For now let's just say that gadgets can be given individual names, and you can specify that an ARexx message containing such a name be sent to LHARCPORT whenever the user clicks on the gadget.

"To guarantee that all messages are handled in a timely fashion, you must clear out queued messages before going back to sleep."

The technique used to handle messages that arrive at LHARCPORT is no different from that used in a C program. Begin with an outer loop that runs until you explicitly break out of it by way of the leave instruction. At the top of the loop, call the ARexx waitpkt() function to go to sleep until a message arrives at LHARCPORT. If a message is already waiting in the message port, then the call to waitpkt() returns immediately; otherwise, it does not return until a message arrives. When a message does arrive, waitpkt() returns and control transfers to the inner (do ff = 1) loop. Here the work of handling the message is accomplished.

The strategy in this section of the program is to keep pulling messages from LHARCPORT until no more messages are pending. This is possible because the getpkt() function lets you poll the port. If a message is queued at the port, then getpkt() returns its address. If no messages are waiting, it returns the string 00000000x. One way to check for a message is to use the ARexx function c2d() to convert this string to a decimal number and compare it to zero. If the c2d(p) is zero, no message exists, so you leave the ff loop. Each time you leave ff, check the value of quitflag to see if it is time to shut down. If quitflag is zero, call waitpkt() to send the program back to sleep until a new message arrives.

It is important to understand just what is meant by a "new message." Each time you call getpkt(), all messages already queued at the port become defined as old, and a subsequent call to waitpkt() ignores the existence of old messages. Therefore, in order to guarantee that all messages are handled in a timely fashion, you must clear out queued messages before going back to sleep.

Assuming that c2d(p) is not zero, use the getarg(p) function to extract the contents of the message. The general syntax of getarg() is:

x = getarg(p,n)

The first argument must be the address of the message obtained from the call to getpkt(); the second argument is an integer that can run from 1 to 15. Using x = getarg(p) without a second argument is equivalent to x = getarg(p,0).

To understand the meaning of the second argument, you have to know that an ARexx message has 16 slots that can contain independent strings. When you use the AddGadget() function, you tell LHARCHOST the information to be loaded into the ARexx message it will send to LHARCPORT. In fact, you can specify that different information be placed in each of the 16 available slots in the ARexx message.

As you will see when we examine the subroutine MakeWindow(), the message associated with a button gadget contains the name of the gadget in its zero slot; all other slots are empty. In the case of a string gadget, the name of the gadget appears in the zero slot, and the first slot contains the current contents of the gadget. A button gadget sends a message to LHARCPORT whenever it is clicked, whereas a string gadget returns a message only when a carriage return is hit after the gadget has been activated for input. As the skeletal listing above shows, the value of the variable command—the string contained in the zero slot of the message—is used to determine what subsequent actions to take.

Note that after a message is received using the getarg() command, it must answered using the reply(packet,rc) function, where packet is the address of the message returned by the call to getpkt(), and rc is a return code, which in this case should always be zero.

Failure to faithfully reply to all packets received can result in disaster. Examination of the skeletal listing shows that when a CLOSEWINDOW command is received—when the user clicks on the window's close gadget—you're performing two operations only. First, you're telling LHARCHOST to close its window by calling CloseWindow(LHARCHOST). Second, you're setting the value of quitflag to 1. This allows the ff=1 loop to continue running until all waiting messages have been replied to.

Note that the full listing does not contain any code to handle commands corresponding to the names of the string gadgets. This is because you read these gadgets at the last moment, before executing a Lharc command, to avoid problems that would result if the user didn't type a carriage return when he finished typing the string. Some sections of the select construct handle commands that require the program to get additional information from the user. To do this, you utilize calls to the postmsg(), request(), and getfile() functions. (The first two functions were covered in my last article; however, the getfile() function is new. See the comments in the full listing for an explanation of its syntax.)

OPENING THE WINDOW

Opening a window and adorning it with gadgets requires only the OpenWindow() and AddGadget() functions. In this

program, you use two slightly different forms of the call to AddGadget(), namely:

```
call AddGadget(hostname,x,y,gadgetname,gadget text,"%d")
```

and

```
call AddGadget
        (hostname,x,y,gadgetname,gadget text,"%d%1%g",length)
```

The first form of the call to AddGadget() is familiar from fastmenu.rexx, but the second is new. In both calls the variables hostname, x, y, gadgetname, and gadget text specify the rexxarplib host to which the command will be sent, the gadget's location in its window, the name to be associated with the gadget, and the default text to appear in the gadget.

In the second call, the %d%1%g string is new. It says that whenever the gadget is activated, the rexxarplib host should forward an ARexx message that contains the name of the gadget in its zero slot and the contents of the string gadget in its first slot. (See the full listing and rexxarplib.doc for a complete specification of the way in which report strings can be constructed.)

The last argument, an integer, serves a dual purpose. First, its existence tells the rexxarplib host that this is to be a string gadget and not a button gadget. Second, it specifies the length of the string gadget. You will also find calls to the SetReqColor(), WindowText(), and SetGadget() functions in this program. These commands were not discussed before, and there are some points concerning their usage that should be made here.

"WindowText() renders text exactly as it appears in the string, including spaces, except that it recognizes a backslash as the instruction to begin a new line."

NEW COMMANDS

SetReqColor(host,pen type,color) tells LHARCHOST which colors to use when rendering gadgets, borders, menus, and so on. Possible pen types are BLOCK, SHADOW, DETAIL, BACKGROUND, PROMPT, BOX, OKAY, and CANCEL. For a four-color Workbench screen, the possible color numbers are 0, 1, 2, and 3. Using SetReqColor() to change these pens can result in interesting effects. In general, however, rexxarplib chooses default values that work well under AmigaDOS 1.3 and with the new 3-D look of OS 2.0. For this reason you only reset the BACKGROUND pen. This must be done because you want to produce a window that is backfilled with a color other than zero.

Note that the rexxarplib host must know all of the requester colors before it renders a window, and the call to SetReqColor() must precede the call to OpenWindow().

There are two techniques for having a rexxarplib host render text in a window. The most flexible method is to use the command Move(hostname,x,y) to define the position at which the text is to begin and Text(hostname,string) to cause the text to be rendered. This method, however, can be slow and unnecessarily complex for our purposes. A simpler method is to use a call to the WindowText() function. The syntax of this call is:

```
call WindowText(hostname,string)
```

WindowText() begins rendering the string at the top of the window and continues until it is done. It renders the text exactly as it appears in the string, including spaces, except that it recognizes a back slash (\) as the instruction to begin a new line. Since the entire string is rendered at once, the call to WindowText() is quite fast. The listing shows that it is fairly easy to construct a string that places text exactly where you want it.

REFRESHING STRING GADGETS

Now I must comment upon the technique you are forced to use to refresh the contents of a string gadget. Unfortunately, it is circuitous. There is no rexxarplib function that lets you change the contents of a string gadget once it has been added to a window. To change the contents of a string gadget, you must first remove the gadget and then add it back. For example, when the user fills in the archive name in the file requester and clicks on OKAY, the string gadget called ARCHIVENAME is updated. This is accomplished by successive calls to RemoveGadget() and Add-Gadget(). Note that, while the call to RemoveGadget() tells LHARCHOST to remove the gadget from its list, it does not remove the gadget imagery from the window. If necessary, this must be done separately, but since you are rendering exactly the same gadget image as before, you can skip this step.

READING STRING GADGETS

Finally, I would like to comment on the subroutine GetVar(), which is called by GetStrings(). This is a general purpose routine that I use in many programs to find the current contents of a string gadget. The routine begins with a call to the ReadGadget(host,gadgetname) function, which tells the rexxarplibhost to read the contents of the indicated string gadget, put those contents into an ARexx message, and send that message back to LHARCPORT. Since you want to process this message within the subroutine, simply call waitpkt() immediately after issuing the call to ReadGadget() and process the message when it arrives. As always, check that you have gotten a valid packet address before calling getpkt(); then reply() to the message.

IN CLOSING

I hope that my articles have served as a useful introduction to some of the possibilities open to you through rexxarplib.library. If you read rexxarplib.doc, you will see that we have only scratched the surface. Things become especially interesting when you start using these tools in conjunction with commercial applications. Give them a try. ■

Marvin Weinstein uses ARexx and REXX extensively in his work at the Stanford Linear Accelerator. Write to him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or contact him on BIX (mweinstein).*

TNT

Technical News and Tools from the Amiga Community.

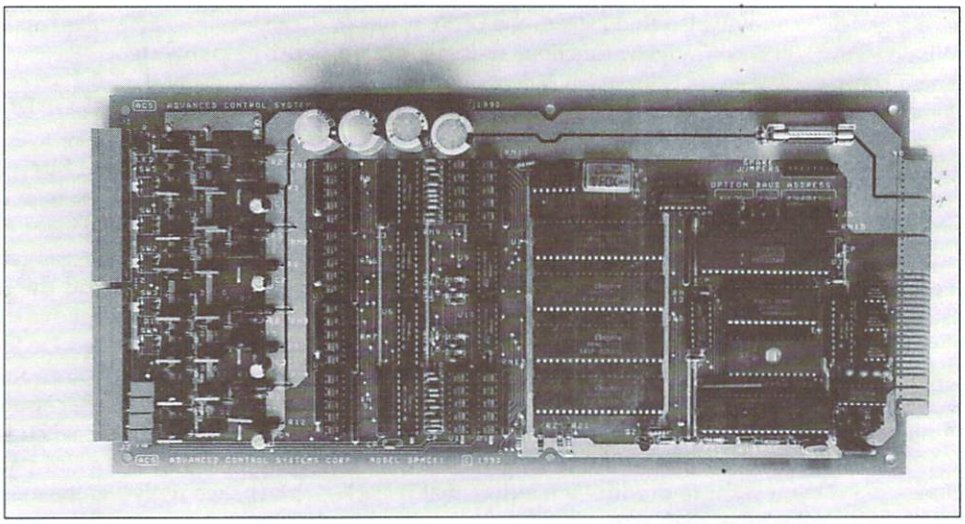
Compiled by Linda Barrett Laflamme

The Quest for Speed

Computer System Associates and RCS Management have joined GVP and Progressive Peripherals & Software in the '040 board race. Besides a 25 MHz 68040, the **40/4 Magnum** from CSA (7564 Trade St., San Diego, CA 92121, 619/566-3911) boasts one to 64 megabytes of 32-bit RAM, a DMA SCSI controller, a parallel port, two serial ports, a 32-bit expansion bus, and room to mount a hard drive. The **Fusion-Forty** from RCS Management (120 McGill St., Montreal, Quebec, Canada H2Y 2E5, 514/871-4924) promises a 25 MHz 68040, four to 32 megabytes of 32-bit RAM, a hardware switch that lets you return to your original processor, asynchronous design, and OS 1.3 compatibility.

Motor Control

Sporting programmable acceleration and deceleration, the **MCB-4 Stepping Motor Controller/Driver Board** can simultaneously control four four-phase stepping motors at two amps per phase. The board connects to your Amiga via the RS-232 port and offers stepping rates of up to 10,000 steps per second and 16.7 million steps per move. It has opto-isolated home and four limit inputs; plus the power section is opto-isolated from the control section in an attempt to reduce noise. Nonvolatile memory for motion control variables and an end-of-motion indicator round out the package. MCB-4 sells for $695 and is available from Advanced Control Systems Corp., Old Mine Rock Way, Hingham, MA 02043, 617/740-0223.



The MCB-4 board controls up to four stepping motors simultaneously.

Faster 500s

Microbotics recently announced a new A500 68030 accelerator—the **VXL-30** (25 MHz, $399; 40 MHz, $629). Based on the 680EC30 chip (which is identical to the 68030, except that the Programmed Memory Management Unit is not available), it allows A500 owners to add an internal 32-bit accelerator to their systems. No one need feel left out; A2000 owners can also use the VXL. To install the VXL, you plug it in the 68000 CPU socket and move the native 68000 to a socket ►

The AmigaWorld Tech Journal Disk

ON DISK

In addition to source and executables for the article examples, you will find:

CATS' Debugging Tools—Enforcer, Mungwall, debug.lib, ddebug.lib

DiskSpeed 4.0—How fast *is* your hard drive?

Libraries & Custom Printer Drivers

This nonbootable disk is divided into two main directories, *Articles* and *Applications*. Articles is organized into subdirectories containing source and executable for all routines and programs discussed in this issue's articles. Rather than condense article titles into cryptic icon names, we named the subdirectories after their associated authors. So, if you want the listing for "101 Methods of Bubble Sorting in BASIC," by Chuck Nicholas, just look for Nicholas, not 101MOBSIB. The remainder of the disk, Applications, is composed of directories containing various programs we thought you'd find helpful. Keep your copies of Arc, Lharc, and Zoo handy; space constraints may have forced us to compress a few files.

Unless otherwise noted in their documentation, the supplied files are freely distributable. Read the fine print carefully, and do not under any circumstances resell them. Do be polite and appreciative: Send the authors shareware contributions if they request it and you like their programs.

Before you rush to your Amiga and pop your disk in, make a copy and store the original in a safe place. Listings provided on-disk are a boon until the disk gets corrupted. Please take a minute now to save yourself hours of frustration later.

If your disk is defective, return to AmigaWorld Tech Journal Disk, Special Products, 80 Elm St., Peterborough, NH 03458 for a replacement.

on the VXL board. This allows you to reboot into 68000 mode when necessary. Additional options for the VXL include a RAM board that holds two or eight megabytes of fast-page mode RAM ($379) and a 68881 or 68882 math coprocessor that can be clocked at speeds up to 60 MHz ($200). For those users who want a PMMU, 68030 versions of the board are available on request. Simply contact Microbotics Inc., 1251 American Parkway, Richardson, TX 75081, 214/437-5330.

In Charge On-Line

A collection of interrelated program modules, **DLG Professional** ($199) takes a new approach to bulletin-board operating systems. Because DLG is built on the Shell and is ARexx compatible, you can incorporate CLI- and ARexx-based programs into DLG's bulletin-board setup. DLG Professional promises to support 65,000 users, 255 user levels, 9999 message areas, 9999 file areas, multiple lines, and conferencing. DLG builds on the standard list of BBS features with the likes of message broadcasting, tagging, bundling, and downloading; off-line reading, and sysop-configurable file-transfer protocols. Plus, it is compatible with FidoNet electronic mail and echomail conferencing protocols, as well as UseNet. For complete details, contact TelePro, 20-1524 Rayner Ave., Saskatoon, Sask., Canada S7N 1Y1, 306/665-3811.

Add a CD-ROM System

Need CD-ROM drive control for your project? Consider **CDROM-FS** ($50 Canadian), a ISO-9660 and HiSierra file system. The Developers Toolkit consists of the cdrom.library of support functions, header and include files, sample source code for calling cdrom.library functions, two stand-alone utility programs, linking library modules for Manx and SAS C, AutoDocs, and sample mountlists. Some of the new features in this release are support for AmigaDOS 2.0 packets, extended attribute records for files and directories, Chinon CDA/CDS/CDX-431 drive support, CDDACtrl audio control, timecode displays for CDDACtrls, and optional audio notification at end of track for CDDACtrls. For a thorough description, contact Canadian Prototype Replicas, PO Box 8, Breslau, Ontario, Canada N0B 1M0, 519/884-4412.

A New Type

Compatible with all Amigas, the **KB-Talker** ($69.95) is a universal keyboard adapter that allows you to connect any PC/AT-compatible keyboard to your Amiga. Completely transparent, it does not require you to make any software changes or additions. A500 owners, however, need a special adapter cable and must make some slight modification to the machine's case. KB-Talker also supports a dual keymapping configuration that can be toggled on the fly, making it a standard PC keyboard for use with Amiga Bridgeboards. Direct your inquires to Co-Tronics Engineering, PO Box 5146, Glendale, AZ 85312-5146, 314/429-2644.

Graphic Improvements

New from GfxBase is the **GDA-1** (512K, $495; 1024K, $649), a Zorro II 16-bit graphics card with an eight-bit display. Resolutions range from as low as 640×480 noninterlaced with 256 colors (out of a palette of 16.7 million) to, for the one megabyte version, 800×600 or 1024×768, each with 256 colors. The display architecture uses chunky pixels, not bitplanes. The memory is contiguous and the card can double as a standard memory card. Current software support includes XWindows for AmigaDOS, and it comes with some basic device drivers, source code, and utilities. A software emulation of the GDA-1 is also available for developers who want to support the card. Current plans call for a December 1991 shipping date. GfxBase Inc. welcomes developer inquires at 1881 Ellwell Dr., Milpitas, CA 95035, 408/262-1469.

What's on the Schedule?

If you or your company has a hot new product on the way, tell us about it. We'll tell the readers. Please send your press releases and announcements to *The Amiga-World Tech Journal*, 80 Elm St., Peterborough, NH 03458, or llaflamme on BIX. ■

CENTAUR SOLUTIONS

COLORBURST

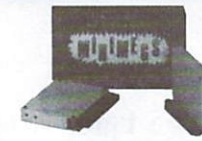
24/48 BIT GRAPHICS ENGINE
The only true 24-Bit video display for all Amigas

- Each and every pixel can be any of 16.8 Million Colors
- Pure, Broadcast Quality RGB Output
- Realtime Image Processing
- Includes 24-Bit Paint Program
- Connects thru Monitor Port to All Amigas
- Compatible with all Amiga Monitors
- High Resolution 768 x 480 pixels (580 PAL)
- Includes 1.5 MB display RAM
- Realtime Horizontal and Vertical Scrolling
- Complex Color Cycling and Video Effects
- All at an Affordable Price

NEW!

MINIMEGS RAM EXPANSION

Affordable 2 MB External RAM
Available for Amiga 500 and 1000
Fully AUTOCONFIG, 100% true FAST RAM
Compact, low profile design
Low power consumption



A Great Value!

Enhanced Unidrive

The Ultimate Drive for your Amiga:

No clicking!
Low Profile Design
Floppy drive port pass-thru
Digital LED track display
Virus Protection switch
Hardware write-protect switch
On/Off switch

FLASH SCSI CONTROLLER

NEW!

The Fastest Controller Available!
Expandable to 16-Bit NON-DMA
Uses Zorro II or III slot.
1 MB ZIP Sockets for user-installable RAM up to 8 MB
Supports Commodore's Rigid Disk Block Structure
8-Bit = 450k/sec transfer rate
16-Bit = >950K/sec transfer rate

8- or 16-Bit NON-DMA SCSI Controller for the Amiga 2000 Series

B.A.D.

The Ultimate Disk Optimizer

- Make your disks fly with B.A.D.!
- Speed disk access time by up to 500%
- Works with floppies AND hard drives
- Support for Virtual Memory and multiple partitions
- Incredible Workbench and CLI performance
- AmigaDOS 2.0 compatible
- Works with the Fast File System
- The MOST popular Amiga utility ever

Mindlink

The hottest, newest, super-powerful modem package

- X-Modem, Y-Modem and Z-Modem protocols.
- Integrated scripting language - create your own modem utilities
- Text Clicking - Just click on a word on the screen to transmit it to another computer!
- Password protection
- Built-in, programmable timer
- Fully multitasking and Workbench 2.0 compatible

PIXOUND

The Musical Graphics Player

Transforms Amiga graphics into music. Load any graphic image (or create one using Pixound's dynamic built-in screen generators), then hear it translated into a rich variety of harmonies and melodies using the Amiga's voices, a MIDI keyboard, or both. You've never seen or heard anything like PIXOUND before.

My Paint

Everybody loves My Paint. Designed for kids but fun for everyone. Includes an animated-icon interface, drawing tools, special effects, multiple palettes, digitized sound effects, 28 pictures to color in and much more! Additional Coloring Book disks available. CDTV Version also available

BLITZ BASIC — The Amiga's fastest BASIC

The ease of BASIC with the speed of Assembly code – Compiled and executable output – Full support for Blits, Sprites, IFF Screens (including HAM), Sampled Sounds, Double-buffered animation and much more!

NEW!

Imagine: A Guided Tour

This extensive video tutorial includes segments on object loading and creation, surface attributes, lighting techniques, texture mapping, animation, 12 and 24-bit comparisons and more. A must-have if you're serious about unleashing your imagination with Imagine.

DCTV: A Guided Tour

This easy-to-follow, comprehensive VHS tutorial will teach you everything you need to know about DCTV! Topics include:

- Installation.
- Using the Video Digitizer.
- Using DCTV with the Video Toaster.
- Using DCTV as a 24-Bit animation display board.

Also includes an interview with DCTV's designer!

Professional Techniques for Deluxe Paint III

The best-selling Amiga video ever. Explore cel animation, 3D perspective, special effects, shortcuts and more. Learn how to get the most out of Deluxe Paint III from artist Jeff Bruette and DPaint's creator, Dan Silva.

Centaur Software

4451-B Redondo Beach Blvd. Lawndale, CA 90260
Phone: 213-542-2226 - FAX: 213-542-9998

Amiga User Interface Style Guide

Rules for the 2.0 look and feel.

By Dan Weiss

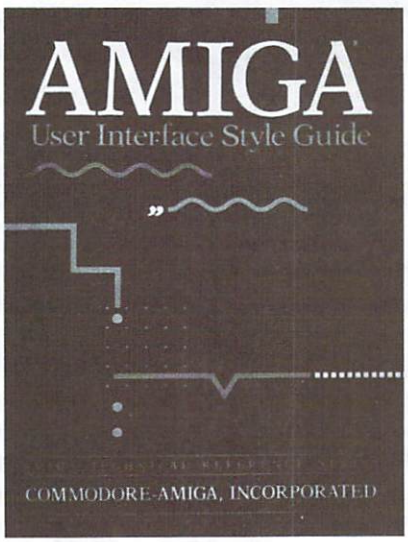
"AT LAST" WAS the first thought that ran through my mind when I saw the *Amiga User Interface Style Guide,* written by Dan Baker, Mark Green, and David Junod of CATS (with the help of many prominent contributors). At last, developers have a definitive guide to help them make some sense of what a "true" Amiga application should look and behave like.

A user-interface guide is not unique to the Amiga. Most commercially available GUIs (graphical user interfaces) have some sort of interface or style guide. The Amiga has been slow in gaining one (there is an eight-page section in the *Amiga ROM Kernel Reference Manual: Libraries & Devices*, but it is vague at best). An interface guide should set a standard for the look and feel of programs that run under the described GUI. Now, in conjunction with the new user interface introduced in Release 2 of the operating system, Commodore has produced such a guide. For the first time, Amiga programmers have a tangible guideline to follow.

The Amiga User Interface Style Guide provides clear explanations suitable for all readers, from the least to the most experienced. Its 200-plus pages cover each facet of the user interface in detail—not only what a button should look like, but also how it should behave under a number of circumstances. This depth sets the book apart from similar publications. It is not simply a description of the Amiga interface, but a unified, exhaustive examination of all its parts. If you want your product to have the Release 2 look, this is where it is defined.

FOR DESIGNERS AND PROGRAMMERS

For the designer creating a new program, the guide lays out a tapestry of ideas. It is important, though, to remember that the book is only a guide. If you need or want to do something different, no one will stop you. Of course, if your approach is really nonstandard, no one may buy your product. Close-up pictures (to the pixel level) of key graphical components allow you to match exactly many facets of the interface that should be



common to all programs (such as the wait cursor). The guide also covers issues of layout and design as they apply to screens and requesters. With this information, you can make sure that your program *looks* like a 2.0 program, but there is more to it than that.

The book also helps programmers make sure their software has the proper *feel*. Before you implement a requester, for example, you should check Chapter Four: Windows and Requesters to see if your requester will act the same way as the standard. Is it draggable? Does it offer the user a safe way out? Or, if you are putting up a requester to notify the user that you were unable load a module or library, are you doing it the best way?

Issues of proper implementation are very important in the guide. For example, when using the newly created cycle gadgets, you must be sure to support standard keyboard shortcuts and determine whether they are modified by the shift key. All major application gadgets, as well as system gadgets, are covered in some detail in Chapter Five: Gadgets.

The guide changes direction with Chapter Eight: The Shell. It addresses the key issues for the parts of the Amiga that are not graphic related. The Shell (8), ARexx (9), and Preferences (12) each has a separate chapter. These, along with The Keyboard (10) and Data Sharing (11), make up the remainder of the instructional part of the book. I would subtitle this section "The Amiga Way."

While the first seven chapters cover material that is more or less common to all GUIs, the final five attempt to extol the virtues of that which is uniquely Amiga. The chapter on ARexx goes farther than any document I have seen in defining a standard set of commands. To date, ARexx support has often been a case of "catch as catch can." The other chapters in this section also offer insight into what Commodore would prefer an Amiga product to be like. Chapter 12: Preferences is perhaps the best in the book at explaining why you should follow the suggested method.

TELL ME MORE

Which brings me to the major failing of the book—justification. For the average reader, being told that long menus are bad is enough. But why are they bad? That's a very important issue. Much research has gone into the design of GUIs, as alluded to in the first chapter, but never referenced. While a fine two-page list of Commodore addresses around the world was provided, no bibliography of further or supporting texts was included. Seems to me that would have filled the 12 blank pages at the end of the book nicely. To make up for this lack, I have included a short list of supplemental reading suggestions on the following page.

As for the design of the book itself, it is the finest of all the Addison-Wesley Amiga technical books to date. The ►

Suggested Additional Reading

The Human Factor
By Richard Rubinstein and Harry Hersh
Digital Press
ISBN 0-932376-44-4
An excellent first book on the subject.

Designing the User Interface
By Ben Shneiderman
Addison-Wesley
ISBN 0-201-16505-8
A treasure trove of information.

The Art of Human Computer Interface Design
Edited by Brenda Laurel
Addison-Wesley
ISBN 0-201-51797-3
A collection of essays heavily slanted towards the Macintosh—interesting in that authors range from Alan Kay to Timothy Leary. □

cover is eye-catching, but not annoying. The layout has an open quality, and is set in comfortable, eye-pleasing, type. Each page has a wide outside margin (great for jotting notes) that sometimes contains hints or short summations, which make scanning for a given section very easy. Including the appropriate chapter number in each page's folio, however, would have made the book a little easier to navigate. The glossary and the index are both excellent, although the index could have been in larger type.

On the whole, I like the guide. Not only does it provide some much needed information, but it is also a pleasure to use. When you get your copy, sit down and read it cover to cover. I did so pool-side, and finished it in no time at all. In the process I found myself taking notes, because I had found answers to several nagging questions. If you are genuinely interested in user-interface design, I suggest you also read the supplemental books listed in the box above. Interesting to note is one line of fine print on the copyright page: "As with all software upgrades, full compatibility, although a goal, cannot be guaranteed, and is in fact unlikely." Oh well, so much for the rule book. ■

Amiga User Interface Style Guide
Addison-Wesley Publishing Company
Reading, MA 01867
617/944-3700
$21.95
ISBN 0-201-57757-7

"Pure" Tricks with SAS/C

By Michael Weiblen

USERS APPRECIATE THE flexibility pure programs offer. By using the RESIDENT command, they have better control over integrating your program into their environment. This article will discuss the advantages of making pure programs resident, which factors determine a program's purity, and a convenient method for creating pure programs using the SAS/C compiler.

PURITY AND RESIDENT

When a user makes a program resident, an image of the program is loaded from disk and added to the Shell's resident list. Later, when the user invokes that program, the Shell first searches to see if the program is on the resident list. If it is, the Shell can skip the time-consuming process of loading the program from disk, and simply begins executing the program image already in memory. When the program finishes executing, the program image remains in memory on the resident list for future reuse.

Contrast this with the invocation of a nonresident program: The program image is loaded from disk, the program executes, and the image is flushed from memory. Clearly, by eliminating the load from disk, a resident program can execute much faster.

The important point here is that every invocation of a resident program will use the same in-memory image. It is vital that no invocation do anything to damage that image. This characteristic is called being serially reusable. Furthermore, given the multitasking capabilities of the Amiga, it is quite possible that a user might want to run several invocations of a resident program simultaneously, so it is equally vital that concurrent invocations not interfere with each other. This is known as being re-entrant. Only programs that are both serially reusable and re-entrant can safely be made resident. Such programs are said to be "pure." Unfortunately, purity doesn't just happen all by itself.

What, then, is necessary to make a program pure? Assuming that the program is not self-modifying (which is a can of worms I'm going to avoid entirely), the factor that distinguishes invocations of a program is its data. Therefore, a pure program must ensure that the data areas of each invocation do not interfere with each other.

WRITING PURE C

In C, data storage can be separated into two classes, local variables and global variables. Local variables, without the static keyword, are allocated by a C function from the task's stack. Because each invocation's task is allocated a separate stack area by the operating system, you can be sure that any data stored on the stack will be private to that task. Local variables, then, cause no problems with a program's purity.

The problem is with how C allocates a program's global variables. Declared outside a function or with the static keyword, global variables are stored as part of the program image, the same image that must be reused and shared among all invocations when the program is made resident. Obviously, a pure program cannot modify its global variables. But what are variables for, except to be modified? The issue of purity, then, boils down to how a program resolves this conflict regarding global variables.

The brute-force method of writing a pure program in C is to avoid using global variables altogether. Each function allocates all its variables locally, either on the stack or from system memory; functions that need to share variables always have to pass around pointers to those variables. Writing a C program using this method is not difficult, but does require a special effort from the beginning. Converting an existing C program—one already using global variables—to this method can be a tedious and cumbersome task.

SAS/C provides an alternative, one that places fewer restrictions on your coding style and is much easier to use when making an existing program pure. All the grunt work of this method is handled for you in a special start-up module, cres.o, which you link into your program instead of the normal start-up module c.o. Many programs can be made pure simply by relinking with cres.o. Other programs, specifically those using some of the more powerful features of the Amiga and version 2.0 of the operating system, may need minor modifications.

Before we look at how cres.o works and what modifications may be necessary to use it, I must say a word about when not to make programs pure. It's tempting to make every program pure, because that gives the user the freedom to make them all resident to improve their performance. If users would never want to make a program resident, however, do not make it pure. A pure version of a program will generally require a little more memory for a solitary invocation than an impure version; it's during the subsequent or simultaneous invocations that the advantages of a resident program emerge. On the flip side, those naive people who try to make everything resident will be quite frustrated when impure programs crash; consider your target users carefully.

CRES.O AT WORK

Cres.o relies on the base-relative method of accessing global variables. Base-relative addressing requires all the global data to be merged into a single data hunk (called near data

Follow these guidelines to make your code safely re-enterant and serially reusable.

in SAS/C's manuals) in the executable file. (This technique is called base-relative because variables are accessed using a 16-bit offset relative to the base address of the data hunk, which is stored in register A4.) The converse of base-relative addressing is absolute (far) addressing, which uses a 32-bit absolute pointer to the variable. While it is quite legal to create a program that uses a combination of absolute and base-relative access to data, this is not usually permitted in a pure program (you may use absolute references to data only if the data is treated as read-only). Cres.o demands that only base-relative addressing be used to access global variables. When you link your program with cres.o, BLINK will display warning messages for any absolute references it detects. Your program is not pure unless it links without warnings.

Now let's follow cres.o in action to see how it makes a program pure. When the program is invoked, cres.o allocates a block of memory and initializes it to 0 to account for the BSS and copies only as much of the data portion of the program image as was initialized at compile time (which is useful in reducing the size of the executable image). For example:

```
init i = 1; /* This data would be copied. */
char array [1000]; /*This data is not copied because it is BSS. */
```

It then loads the address of that private data area into register A4 and begins executing your functions. Effectively, every invocation starts with a fresh copy of the global data area, which it accesses relative to its private A4 base address. Those "global" variables are then actually global to the invocation, not the image. When your functions finish, control returns to cres.o, which deallocates the data area and exits to the OS.

The importance of avoiding absolute references to global variables should now be apparent. Where a data area is located depends on which invocation it belongs to. Absolute references, by their very nature, are not capable of this adjustment. (In fact, absolute references really access the data portion of the resident image. We definitely wouldn't want to disturb that!) The problem is that absolute references can slip into your program in several subtle ways.

WARNINGS

Here is a list of trouble spots and methods to avoid them.

- Never use compiler flag –b0.

The purpose of this flag is to force absolute addressing of global variables; the far keyword does the same thing, as absolute addressing is the same as far.

- Avoid linking with amiga.lib if possible.

Amiga.lib contains lots of stuff, including loads of absolute references. Particular culprits are the system library stub functions. Amiga library functions are accessed as offsets relative to the library's base pointer. All the details about how to access library functions are encased in these small stub functions. The problem is that these stub functions make absolute references to the global variables that contain their respective library base pointers.

As an alternative, SAS/C supports pragmas, which are commands to the compiler describing everything necessary to directly access the library functions. Essentially, pragmas give the compiler all the information necessary to generate stub functions on demand. The advantage is that pragmas will use the addressing mode in effect when the source file is compiled, which for pure programs must be base-relative addressing.

Some functions, however, don't have pragmas. For example, the "varargs" form of several of the new OS 2.0 functions do not exist in a library, hence they don't have pragmas. Actually they are small "wrapper" functions in amiga.lib that adjust parameters before calling a related library function. The solution is simple: Write your own version of these wrapper functions. (The Weiblen drawer of the companion disk includes a sample of a replacement wrapper function.)

Sometimes you just can't avoid linking with amiga.lib; it does contain a lot of useful (and pure) routines. Just make sure that BLINK does not report any absolute-reference warnings, such as:

```
Warning! Absolute reference to _SysBase
    module:      file: lib:amiga.lib
```

- Never use the standard directives for retrieving the A4 global data segment pointer.

These directives (which include the compiler flag -y, the function geta4(), and the __saveds keyword) load the base address of the global data area into register A4. They are used when functions requiring access to global variables are run under another task's context. Specific examples include spawned tasks, interrupts, and hooks. These directives don't work in pure programs because they have no way of knowing where each invocation's global data segment is located. The Weiblen drawer contains a set of simple replacements for these directives, along with demo programs illustrating their use.

By following these guidelines and linking with cres.o, you should be able to generate pure programs quite easily. ■

Michael Weiblen is an engineer at IMSATT Corp., the creators of AmigaVision. Contact him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or on BIX (mweiblen).*

Inside MIDI

A hardware and software tour of the music standard.

By Mike Harris

OVER THE PAST decade, an overabundance of standards has appeared in the computer industry. Few, if any, have had the universal acceptance of the Musical Instrument Digital Interface (MIDI). Perhaps this is because of the unique nature of its origin. While most industry specifications originate in ANSI, ISO, or IEEE committees, MIDI rose from informal meetings of leading electronic keyboard manufacturers. Sequential, Roland, Yamaha, Korg, and Kawai met in Tokyo in August 1983 and finalized the MIDI 1.0 specification. This version remains in use with minor additions and changes to the software. The 1.0 spec consists of the hardware required and the data format to be used, as you will see.

THE HARDWARE

A MIDI device uses asynchronous serial communication to transfer data between equipment. The rate of transfer is 31.25 Kilobaud with a specified 1% tolerance. The standard 8N1 protocol yields a speed of one byte every 320 microseconds (10 bits/31,250 bps). IN and OUT ports are used for I/O, while a THRU port provides a copy of the signal entering via the IN port for a secondary device. Included in the spec is a diagram of a sample circuit as a reference for implementation; however, it is not the required method.

Rather than using a bipolar voltage as a signal as RS-232 does, MIDI uses a 5 mA current loop. At the receiving end of the signal, an opto-isolator is required (the reason for the current loop). The isolator must have rise and fall times under 2 microseconds and need less than 5 mA to be turned on. A Sharp PC-900 and HP 6N138 (see Figure 2) are the recommended parts, but many high-speed opto-isolators are satisfactory. The reason for the opto-isolator is straightforward: to isolate the ground between equipment. Ground loops result in an audible hum in some audio gear and thus must be avoided.

The IN, OUT and THRU ports of a MIDI device use five-pin female DIN connectors. Remember that all equipment does not always need all three ports and also that multiple INs and OUTs are common. Connectors are another area where ground loops can occur. You will notice that OUT and THRU have pin 2 grounded while IN does not (see Figure 1). This allows the shielding to be grounded on the cables and still not cause a ground interconnection. The spec calls for cabling under 50 feet long and made of twisted-pair wire with the shielding connected to the male connector's pin 2 at both ends. As long as the shells of the female connectors are not tied to ground, a ground loop will not occur.

THE DATA FORMAT

A complete discussion of what is needed to implement MIDI software deserves its own article; here we will cover the

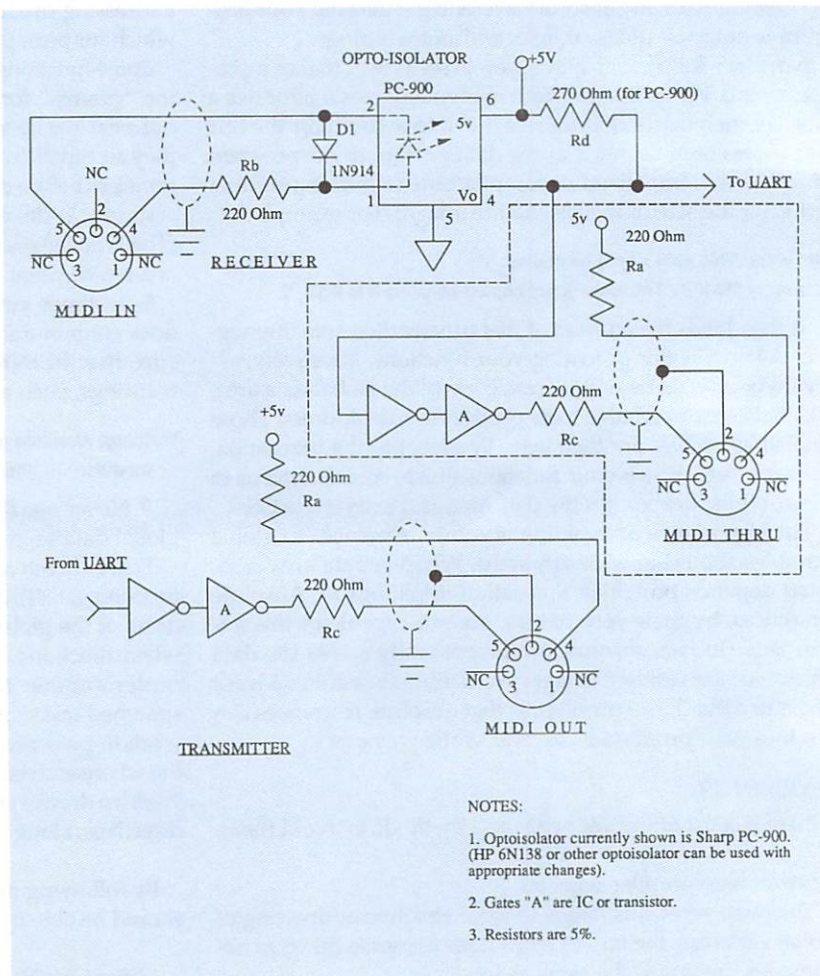


Figure 1. The MIDI hardware standard.

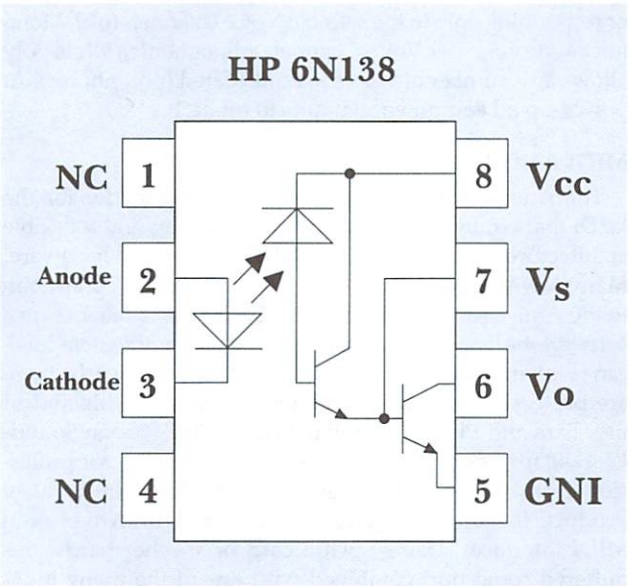
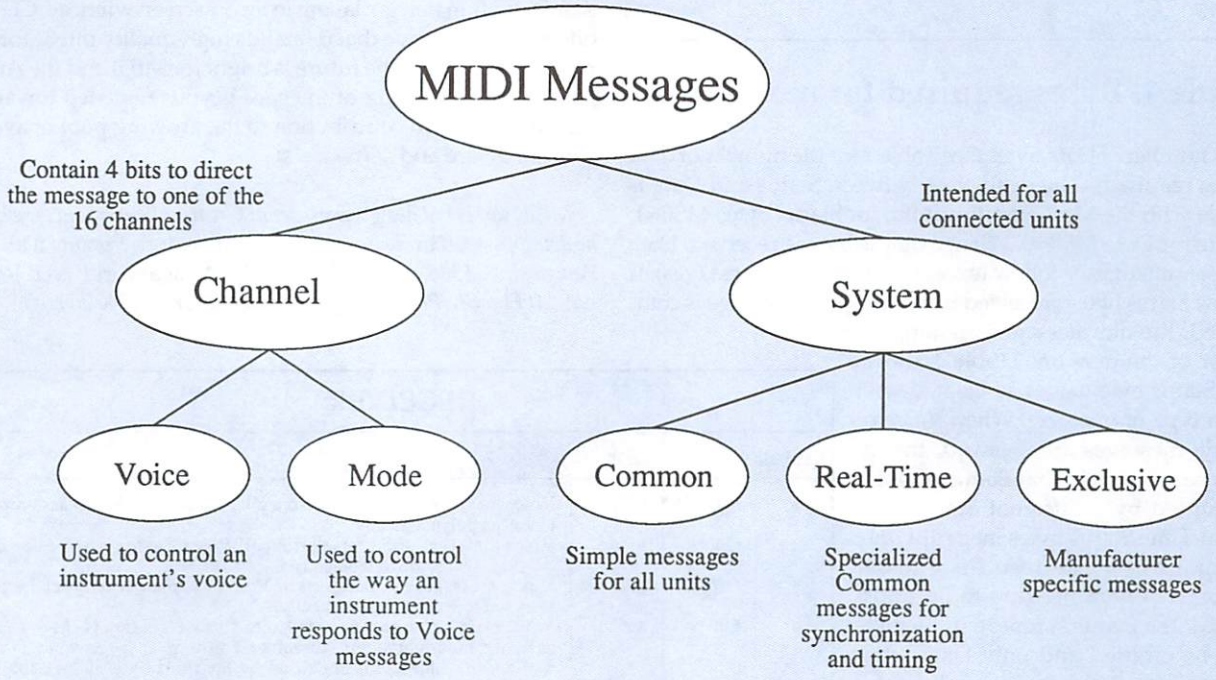


Figure 2. The HP 6N138 opto-isolator.

general concepts of the MIDI format only. If you require more information, you can obtain a copy of the MIDI specification for $38 from the International MIDI Association, 5316 West 57th St., Los Angeles, CA 90056, 213/649-6434. The standard MIDI file format addendum is an additional $5.

MIDI messages are broken down into two main categories, Channel and System, which in turn are broken down into subcategories. Figure 3 illustrates the heirarchy and describes the message types.

All MIDI communication consists of a Status byte followed ►



Channel Messages: Contain four bits to direct the message to one of the 16 channels. Only units whose channel number matches the channel encoded in the Status byte are of interest.
Voice: Used to control an instrument's voices.
Mode: Used to control the way an instrument responds to Voice messages.

System Messages: Intended for all connected units.
Common: Simple messages for communicating with all units.
Real-Time: Specialized Common messages used for synchronization and timing.
Exclusive: Manufacturer-specific messages made up of a manufacturer's MIDI ID number and a variable number of Data bytes.

Figure 3. MIDI message organization.

Status D7----D0	# of Data Bytes	Description
Channel Voice Messages		
1000nnnn	2	Note Off event
1001nnnn	2	Note On event (velocity=0: Note Off)
1010nnnn	2	Polyphonic key pressure/after touch
1011nnnn	2	Control change
1100nnnn	1	Program change
1101nnnn	1	Channel pressure/after touch
1110nnnn	2	Pitch wheel change
Channel Mode Messages		
1011nnnn	2	Selects Channel Mode
System Messages		
11110000	*****	System Exclusive
11110sss	0 to 2	System Common
11111ttt	0	System Real Time

NOTES:

nnnn: N-1, where N = Channel #, i.e. 0000 is Channel 1. 0001 is Channel2. . . . 1111 is Channel 16.

*****: 0iiiiii, data, ..., EOX

iiiiii: Identification

sss: 1 to 7

ttt: 0 to 7

Table 1. Bytes required for messages.

by a number of Data bytes (see Table 1 for the number of data bytes required). The distinction between Status and Data is made with the Most Significant Bit; for Status bytes MSB=1, while for Data MSB=0. When a Status byte is received, Data bytes immediately follow (except for Real-Time messages). If a new Status byte is received before the prior message is completed, the old message is ignored in favor of the new one. Table 1 shows the Status byte values associated with each type of message. When Voice or Mode messages are received, the interface remains in that Status until interrupted by a different Status byte (Real-Time Status bytes interrupt only temporarily). The result is a lesser amount of data needing to be transferred. If a Status is repeated, the byte can be omitted and only Data bytes can be sent, which is extremely useful when transmitting long strings of data. Any unimplemented or undefined Status bytes and subsequent Data bytes are ignored.

Each MIDI transmitter or receiver can be in one of four Channel modes for use in voice assignment, as Table 2 illustrates. Each unit is set with Omni on or off and set to Poly or Mono. Omni refers to the equipment responding to all Voice Channels (on) without discrimination or responding only to the selected Voice Channels (off). Mono forces one voice per Voice Channel (monophonic), while Poly allows any number of voices to be allocated (polyphonic). At power-up all equipment defaults to mode 1.

RECEIVER			
Mode		Omni	
1	On	Poly	Voice messages are received from all Voice Channels and assigned to voices polyphonically.
2	On	Mono	Voice messages are received from all Voice Channels and control only one voice, monophonically.
3	Off	Poly	Voice messages are received in Voice Channel N only, and are assigned to voices polyphonically.
4	Off	Mono	Voice messages are received in Voice Channels N thru N+M-1, and assigned monophonically to voices 1 thru M, respectively. The number of voices M is specified by the third byte of the Mono Mode Message.
TRANSMITTER			
Mode		Omni	
1	On	Poly	All voice messages are transmitted in Channel N.
2	On	Mono	Voice messages for one voice are sent in Channel N.
3	Off	Poly	Voice messages for all voices are sent in Channel N.
4	Off	Mono	Voice messages for voices 1 thru M are transmitted in Voice Channels N thru N+M-1, respectively, (Single voice per chennel).

Table 2. Possible MIDI channel modes.

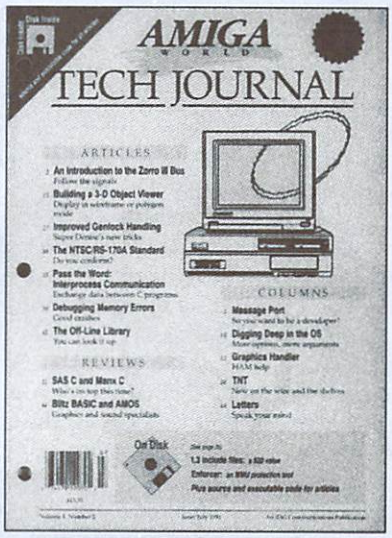
MIDI AND THE AMIGA

The Amiga has always been a strong contender for the MIDI market. Its offering of true multitasking and a flexible architecture lends itself well to MIDI software and hardware. Many low-cost MIDI interfaces are available for connection to the Amiga serial port. One drawback is their reliance on a software-buffered serial port that, under heavy system load, can result in lost or incorrectly received data. Currently, there are professional-quality programs readily available and in use. Bars and Pipes Professional (from Blue Ribbon SoundWorks), for example, offers everything necessary for professional applications. At the time of this writing, Great Valley Products is working on a dual serial card with two built-in MIDI interfaces. Using such a card or another hardware-buffered serial port combined with one of the many high-quality MIDI programs available will result in a sound MIDI solution.

With the Amiga's growing acceptance as a multimedia platform, MIDI can only contribute to its popularity. Soon, products may be available that will allow you to compose a score for a film that is playing in an on-screen window. CDTV offers another venue that demands high-quality music for its programs. Overall, the future is bright for MIDI and the Amiga. An understanding of the spec is your first step towards making your own contribution to the growing pool of available hardware and software. ■

Mike Harris is a design engineer at Great Valley Products, where he develops new hardware and ASICs. He has been working with the Amiga since 1986. Contact him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or on BIX (harris).*

April-May 1991 *Premiere*



June/July 1991

August/September 1991

MISSING VALUABLE ISSUES?

Send for your missing back issues and complete your

AMIGA WORLD TECH JOURNAL

library today!

Call toll free or mail this coupon today!

1-800-343-0728

Mail order to: AmigaWorld Tech Journal Back Issues
P.O. Box 802, 80 Elm Street
Peterborough, NH 03458
800-343-0728/603-924-0100

___Apr/May 91 *Premiere*
___Jun/Jul 91
___Aug/Sep 91

I have checked ___back issues x $15.95	$_____
California orders add 6.25% sales tax	$_____
Canadian orders add 7% GST	$_____
(GST reg. # 126038405)	
Add postage/handling:	
U.S. surface orders - $1	$_____
Canadian surface - $2	$_____
Canadian air mail - $3	$_____
Foreign surface - $3	$_____
Foreign air mail - $7	$_____
Total Enclosed	$_____

___Check/money order enclosed
___Charge my:
___ Mastercard ___Visa ___American Express ___Discover

Card #_____ Exp·_____

Signature _____

Name _____

Address _____

City, State, Zip _____

TJSK1

Designing a Device Driver

By Dan Babcock

EMBEDDING DEVICE-RELATED code in your final application at first may seem less painful than defining a driver for your unique widget, but it limits the options later. You and other programmers will have to rewrite the instructions in each piece of software that plans to support it. By defining a driver, you simplify future work for yourself and other programmers. To make it even easier, chances are you may never need to write a unique driver; emulating the programmer interface of one of the established system drivers—most commonly trackdisk.device, serial.device, or parallel.device—usually does the trick. Existing software will work with your new, similar device—for example, a hard-drive controller—without alteration.

Before we examine the basics of crafting a driver, note that I assume you have some knowledge of device I/O from the application's point of view.

A DEVICE IS A LIBRARY

Believe it or not, although libraries and devices seem radically different from an application's point of view, they share the same overall structure. A library consists of a jump table that the program uses to access the provided routines, a global data structure that the library's routines use as they wish, and a minimum of three standard routines: Open(), Close(), and Expunge(). A device shares the same form, and adds two more special routines: BeginIO() and AbortIO(). Once you understand the function of these two routines, you've got it!

BEGINIO()

BeginIO() is the workhorse entry point in a device. All input and output requests from the user result in a call to BeginIO(). The BeginIO() vector may be called directly by the user, but more commonly it is called as a result of SendIO() or DoIO(). Very little work is performed by SendIO() or DoIO(); they do little more than call the BeginIO() vector. (An exact description of SendIO() and DoIO() is provided later.) Passed to BeginIO() are a pointer to an IORequest in A1 (containing the command number and parameters) and a pointer to the device-base pointer in A6 (just like a library). The BeginIO() routine examines the command code (IO_COMMAND) and determines what, if anything, should be done. The only mandatory features of the BeginIO() routines are that they must call ReplyMsg() for the IORequest if the quick bit is zero and must set the LN_TYPE of the IORequest to NT_MESSAGE. (The latter requirement is an obscure bug fix.) Other than that, the implementation of BeginIO() is left to your imagination.

Implementation would be trivial if not for a couple of issues. BeginIO() may be concurrently called from many tasks. The routines in a driver that manipulate the hardware are almost never reentrant, however, so this is a major issue. Second, by convention BeginIO() is expected to not take a "long time" to complete. A "long time" is really a duration that depends on an external event that may or may not occur in a known period. The reason is that applications expect—and, in fact, sometimes depend on—asynchronous operation when they call SendIO(). In this case, BeginIO() does not actually perform the I/O, but merely passes the request to an independent task for further processing.

The most common solution to these problems is to set up an independent task (or tasks) associated with the driver. BeginIO() simply passes the I/O request to the proper task and returns. This is usually accomplished by using PutMsg() to send the IORequest to a message port associated with the task. PutMsg() uses the linked-list fields at the start of an IORequest structure (conveniently provided for this purpose) to attach the IORequest to the message port, and signals the task to wake it from its dormant state. The task then calls GetMsg() to access the IORequest at the top of the list (and unlink it) and proceeds to execute the requested command. This arrangement neatly solves the reentrancy problem, provides for asynchronous operation, and serializes incoming requests.

This is a lot of work for the device, but it benefits all applications: Rather than an application being forced to spawn a plethora of tasks to handle I/O, the application simply uses SendIO() to achieve concurrency. The amount of work this saves all application writers makes device drivers worthwhile. In fact, this is almost the entire point of having device drivers, as opposed to simple libraries of functions.

Occasionally, you can take another, simpler, approach to writing BeginIO(). Some drivers do not require asynchronous operation to be useful; it is only a frill. In this case, you may choose to dispense with the task business and do everything within BeginIO(). That leaves only the problem of multiple tasks calling BeginIO() concurrently. A simple semaphore suffices to solve this: BeginIO() calls ObtainSemaphore() to do its work, and then finishes by calling ReleaseSemaphore(). If the BeginIO() routine is busy when it is called, the task that called BeginIO() will sleep until the semaphore is released by a call to ReleaseSemaphore(). The code savings of this approach over the task method is considerable.

Remember, however, that some types of drivers—such as serial and disk drivers—absolutely must support asynchronous I/O. Disk drivers are a good example of how a multitasking OS can get things done while slow hardware is working. File

Why continually include specific device instructions in your programs when you can call a driver that does the work for you?

systems can send out asynchronous requests for blocks and process the next block while the previous one is fetched. In such cases, the simpler approach is not an option.

ABORTIO()

The other device-unique entry point, AbortIO(), is passed a pointer to an IORequest in A1 and the device-base pointer in A6, and is expected to attempt to make the I/O job abort and return more quickly than usual. If successful, AbortIO() returns zero in D0 and stores IOERR_ABORTED in IO_ERROR. For some devices (as for disks), aborting is not critical and may be ignored. For others (such as the serial device), the AbortIO() routine is very important and must be handled properly.

To do so requires a bit of advanced planning. Every time the driver task (or tasks) calls the Exec Wait() function to wait for an event, it should simultaneously wait for a special abort signal. To force an early exit, the AbortIO() routine then can send this abort signal to the relevant task. The I/O operation that is to be aborted might not actually be active at the time AbortIO() is called, however. There are two such cases: either the IORequest has already finished or the IORequest is sitting in a queue waiting to be processed. Note that it is illegal for an application to abort an I/O request that has not been initiated yet, so that scenario is not a concern. If the LN_TYPE field of the IORequest is NT_REPLYMSG, then the I/O has completed, and ReplyMsg() has processed the request. The AbortIO() routine should simply exit. To determine whether the request is currently being processed, AbortIO() simply compares the IORequest pointer passed to AbortIO() with the current IORequest, which should be maintained in the task code for this purpose. This field should be set to zero when no I/O requests are being processed to avoid confusion and to act as a task-busy flag. If the IORequest is not active, but waiting to be processed, then the IORequest may be "defused" by setting a special "ignore" flag in the IO_FLAGS byte of the IORequest; the upper four bits are reserved for the driver's use. When the task fetches the IORequest from the queue (message port), it can test the ignore flag; if it is set, ReplyMsg(IORequest) is executed and no further action is needed.

Implementing the AbortIO() feature can be quite tricky. If your driver requires an abort feature, take a close look at the AbortIO() routine in the example serial driver provided in the accompanying disk's Babcock drawer.

Discussions of the other standard entry points—Open(), Close(), and Expunge()—follow. Although the descriptions generally apply to the library versions of these routines, some of the details are different. These routines, as well as BeginIO() and AbortIO(), follow the usual Exec register conventions: D0, D1, A0, and A1 are scratch; all other registers (except status) must be preserved.

OPEN()

Called by OpenDevice(), the Open() routine receives the device-base pointer in A6 (as usual) plus the OpenDevice() parameters: the IORequest pointer in A1, the unit number in D0, and the flags in D1. Multitasking is disabled by Exec before calling Open(). What the Open() routine does is entirely up to the device. Usually it sets up for a particular unit by creating a unit task (or tasks), setting up a unit-specific data structure, initializing unit-specific hardware registers, and so on.

The driver can keep track of which unit goes with which IORequest by storing a pointer to the unit-specific data structure (or any other convenient indicator) in the IO_UNIT field of the IORequest provided for this purpose. Applications do not use IO_UNIT. The OpenDevice() routine automatically stores the device-base pointer in the IO_DEVICE field of the IORequest, for the later use of the application, Exec, and driver.

CLOSE()

The Close() routine is called from CloseDevice() with the IORequest in A1. Multitasking is disabled by Exec before calling Close(). Usually, Close() checks for outstanding Open() calls on this unit. If there are none, it releases various unit-specific resources. In addition it might choose to call Expunge() (see below) if there are no openers for the entire device. (Expunge () is never called if your opencount is nonzero.) If the device should be unloaded from memory, Close() returns the segment list (as given to the initialization routine; see below); otherwise it returns zero in D0.

Ramlib (used when the driver is first called) uses semaphores to prevent multithreading Open() and Close(); Init/Forbid() is a side effect. Expunge() is called from the memory allocator AllocMem() with only Forbid().

EXPUNGE()

The Expunge() routine of a device or library is called by AllocMem under emergency low-memory situations. If possible (nothing is currently using the device), the device should deallocate all memory buffers, close anything that needs to be closed, and generally clean up. The device is removed from memory if Expunge() returns the segment list in D0; otherwise Expunge() should return zero in D0. Multitasking is disabled by Exec during the Expunge() routine. ►

Note, however, that drivers should deallocate and release important resources on the last Close(), not Expunge(). Do not pattern your device after the bug that allows serial.device to hold onto miscellaneous resources until Expunge().

THE BIG PICTURE

Up to now the discussion has been from the device's point of view. To put it all together, examine the application-Exec-device interaction in its entirety:

Synchronous I/O

1. The application calls Exec DoIO() with the IORequest pointer in A1.
2. DoIO() sets the IO_QUICK bit of IO_FLAGS by moving 1 into IO_FLAGS.
3. DoIO() loads IO_DEVICE into A6 and calls the device's BeginIO() vector.
4. The BeginIO() routine executes the command (in IO_COMMAND) immediately or sends the request to a task and returns. In the latter case, BeginIO() clears the IO_QUICK bit.
5. DoIO() calls WaitIO().
6. WaitIO() checks the IO_QUICK bit: if it is set, WaitIO() loads the error code into D0 and returns immediately; otherwise it calls Wait() until it receives the reply port's signal bit, unlinks the IORequest from the reply port's message queue, and returns with the error code (taken from IO_ERROR) in D0. (The signals may remain set!)

Asynchronous I/O

1. The application calls Exec SendIO() with the IORequest pointer in A1.
2. SendIO() clears the IO_FLAGS byte (with the intent to clear IO_QUICK).
3. SendIO() loads IO_DEVICE into A6 and calls the device's BeginIO() vector.
4. The BeginIO() routine should merely send the request to an associated task and return, if the request will take some time to satisfy. It may optionally perform the request right away.
5. SendIO() returns.

In all cases the driver should call ReplyMsg(IORequest) if and only if the IO_QUICK bit is clear. In addition, if the driver does not want to complete a given request immediately, it must clear the QUICK bit before returning from BeginIO(). (The associated task should perform a ReplyMsg() when finished.) Note that it is always safe to clear the QUICK bit, but never safe to set it.

HOW A DEVICE IS INITIALIZED

The device jump table and other structures necessary to make a device functional do not appear by magic; something has to create them. Fortunately, this is not the responsibility of the device driver. The only requirement is that the device begin with a RomTag structure. If the driver is present in the DEVS: path, then AmigaDOS will (in OpenDevice()) load the driver and call InitResident() to initialize the device (or library). The RomTag and related structures may be directly copied from one of the sample drivers without much complication, so I won't describe them here. The most interesting part of this process is that an initialization routine is called in the driver. The initialization routine is a very convenient place to perform one-time-only set-up tasks. The routine is called with the device-base pointer in D0 and the AmigaDOS segment list in A0. The initialization routine is responsible for saving the segment list for the later use of Close() or Expunge() (see above). If all goes well, the routine should return the device pointer in D0, or return zero in D0 to indicate an error. Note that the initialization routine follows the usual Exec register convention.

DEBUGGING

Unfortunately, there are no truly adequate tools available for debugging complex drivers (ignoring very expensive hardware solutions). One useful technique is to output debugging text via the internal serial port. See the PUTDEBUG macro in the serial driver for an example. PUTDEBUG accesses the hardware directly, so it may be used anywhere, including interrupt routines. The downside is that it destroys the normal timing of events, hiding bugs and precluding some tests. In the end, there is no substitute for reading the source code and knowing that it must work. This ideal is often thwarted in the real world, however, by cryptic or incomplete documentation for the hardware that the driver is controlling. Patience is definitely a virtue when debugging drivers!

In the accompanying disk's Babcock drawer, you will find two example drivers written in assembly language, as is typical of most device drivers. RAMDISK.ASM acts as a simple RAM disk and represents a "minimal" device driver. NEWSER.ASM is a full-blown serial driver for the Rockwell 65C52, demonstrating all the intricacies of a typical "real" driver. Another good guide is Commodore's example driver printed in the *Amiga ROM Kernel Reference Manual: Libraries & Devices*; the comments it contains represent the official guide to writing drivers. I strongly encourage you to examine these examples, as they contain a wealth of useful information.

INSIDE THE RAM-DISK DRIVER

A RAM disk is good first example because it can be greatly simplified. In this sample, the initialization routine allocates a buffer of 880K (which you may change to any value) and stores the pointer in the device-base structure. Open(), Close(), and Expunge() do essentially nothing; only one unit is supported, so the unit number is not checked. The device is never removed from memory. The BeginIO() routine does not use tasks or semaphores, because they are not needed for a RAM disk. It accepts two commands, CMD_READ and CMD_WRITE. It returns all other commands with IO_ACTUAL=0 without doing anything. AbortIO() does nothing, because there is no way or reason to abort a request. All in all, a RAM disk is not very representative of a "real" driver, but it is a very nonthreatening introduction to drivers and useful in its own right.

INSIDE THE SERIAL DRIVER

The example serial driver is a real-world example of a device driver. It represents close to the worst case in terms of complexity. This serial driver is unusual in that it uses two tasks per unit—one dedicated to reading, the other to writing, because software expects to be able to read and write concurrently. It would be more consistent if the application were required to open separate units for reading and writing, but this arrangement works fine. The BeginIO() routine examines the IO_COMMAND field of the IORequest: if it is CMD_READ, the request is directed to the read task. If Be-

Continued on p. 63

SAVE $5 NOW!

SUGGESTED RETAIL: ~~$24.95~~ now only $19.95 ea.

FOLLOW-UP TO OUR BEST SELLER!

AMIGA ANIMATION VOLUME TWO!

...Selected from hundreds of incredible works.

In response to the clamor for another videotape featuring Amiga animations, the Editorial Staff of AmigaWorld has created ANIMATION VIDEO, VOLUME TWO. AmigaWorld sponsored another contest soliciting entries from talented Amiga animators. The Editors sifted through hundreds of submissions and countless hours of animation clips to select the very best in animated art. The result is a videotape with scintillating animations, showcasing the efforts and talents of Amiga enthusiasts.

ANIMATION VIDEO, VOLUME ONE was a best-selling video, containing commercially broadcast and award-winning work. The second volume is even more exciting, due to such innovative animation programs as Sculpt-Animate 4D, LightWave 3D, Turbo Silver, Imagine and Deluxe Paint III. The animations on this video will impress you with technical brilliance and delight you with imaginative plots. You'll be thoroughly entertained as you absorb new animation techniques and ideas. Whether you just brought your Amiga home from the store or you have created your own animation art before, you'll want to add ANIMATION VIDEO, VOLUME TWO to your Amiga video collection!

1-800-343-0728

CALL TOLL FREE or mail this coupon.

YES! I am eager to become an expert! Please send me the following videos:

❑ Animation Video, Vol. Two	~~$24.95~~	*$19.95*
❑ Animation Video, Vol. One	~~$19.95~~	*$14.95*
❑ Hot Rod Your Amiga	~~$24.95~~	*$19.95*
❑ NewTek's Video Toaster™	~~$24.95~~	*$19.95*
❑ Desktop Video, Vol. One	~~$29.95~~	*$24.95*
❑ Amiga Graphics, Vol. One	~~$29.95~~	*$24.95*
❑ The Musical Amiga	~~$29.95~~	*$24.95*
❑ The Amiga Primer	~~$29.95~~	*$24.95*

❑ Check/Money Order ❑ MasterCard ❑ VISA ❑ AmEx

Make checks payable to TechMedia Video. ❑ Discover

Please include $2.95 shipping & handling for one video, $5.00 for two or more.

Canadian orders add 7% GST (GST reg. #126038405) **Total Amt. $** _____

Acct. # _____ Exp. Date _____

Signature _____

Name _____

Address _____

City/State/Zip _____

TECHMEDIA VIDEO

PO Box 802, 80 Elm Street, Peterborough, NH 03458 603-924-0100

An IDG Communications Company

Available in VHS only. Please allow 4–6 weeks for delivery. Foreign orders: add $7.50 for airmail delivery; $18 for two or more videos. Payment must be made in U.S. funds drawn on U.S. banks. TechMedia Video is the licensed North American distributor of AmigaWorld Videos.

Spawning Tasks

By Steve Krueger

YOUR PROGRAMS CAN launch new processes either synchronously or asynchronously. Synchronous creation is used most often when a program needs to execute a second program and wait for it to finish before continuing. The standard make utility is an example of this type of program. Synchronous processes do not take full advantage of the multitasking abilities of AmigaDOS. In many situations, you might want one process performing a task in the background while another responds to user input—in other words, tasks executing ascynchronously. As synchronous processes are the simplest to create, we'll examine them first.

The system call Execute() is the most common way to implement a new process synchronously. Execute() takes three parameters—a command string, an input file handle, and an output file handle—and creates a new CLI that executes the specified commands as if they had been typed in at the Shell prompt. Once the command or commands are executed, the new CLI terminates and the program that called Execute() continues to run. For example, to use this call to display a directory, you would use the following:

```
Execute("dir", NULL, Output());
```

This executes the command DIR and displays its output to the output file handle, Output(). The call will not work if the program is run from Workbench because Output() does not exist. If the output file handle is NULL, the output goes to the current window. Once again, there is no current window under Workbench.

To open a new window and send the output to it, use the code fragment below:

```
fh = Open("CON:", MODE_OLDFILE);
Execute("dir", NULL, fh);
```

If the second parameter (the input file handle) is not NULL, then the CLI reads from the input file handle after the command string is executed. This continues until the end of the file is encountered. You can use this feature to execute a script file from within a program. If the command string is NULL, input is read from the input file handle immediately.

The Execute() routine has a couple of limitations. When a process run from Workbench calls Execute(), the routine has no default input file handle, and it inherits no path. You must provide an input file handle when running under Workbench. This can be accomplished by calling Open() for NIL:. The path is optional. To provide one, you must create a fake CLI environment and copy the path from the Workbench process. A second drawback to Execute() is that it does not return the return code of the command it executes. Commodore added a new routine, System(), to Amiga OS 2.0 to remedy this situation.

The System() routine is similar to Execute(), but has a few significant differences. It can execute a program either synchronously or asynchronously, and it will not read from the input file handle. The return from System() is –1 if the command could not be executed, otherwise it is the return code for the program. System() takes two parameters: a command string, and a pointer to a tag list. (See "Digging Deep in the OS," p. 8 for an introduction to tags and tag lists.) In the tag list, you can specify the input file handle, output file handle, synchronous or asynchronous execution, and the type of Shell to be used. For more information, refer to dos/dostags.h. Remember, System() is available only under version 2.0 of the operating system.

Both System() and Execute() search the resident list for the specified command. There is no other documented method of executing commands from the resident list.

ASYNCHRONISITY

The setup required to create a new process asynchronously is more involved than synchronous creation. The table below outlines the steps that the parent (original) and child (spawned) processes follow for specific periods of time.

Parent	Child
Set up fake seglist	***
Call CreateProc()	***
Set up start-up message	Wait for start-up message
Send start-up message	Gather needed info from startup
Continue executing	Execute user function
Wait for reply to start-up	Reply to start-up message and terminate
Free child resources	***

*** indicates that the child process does not exist at this time.

The system routine that does most of the work is CreateProc(). CreateProc() takes four parameters—the process name, a seglist, the priority, and the stack size. The process name is a NULL-terminated string that need not be unique. The priority is an integer value specifying the priority at which the new process will run. Size of the new process' stack is specified in LONGWORDS. The seglist is a BPTR pointing to the first code hunk that will be executed. You can obtain a seglist by calling LoadSeg() or create a fake one. Use LoadSeg() if the new process is going to execute a new program. If the new process is to run in the same code as the current program, you must create a fake seglist. Let's try the latter method.

Follow these steps to easily create sychronous and asynchronous processes.

Commodore's AutoDocs for CreateProc() suggest the following code to create a fake seglist:

```
ds.l   0    ;Align to longword
DC.L   16   ;Segment "length" (faked)
DC.L   0    ;Pointer to next segment
...start of code...
```

The example provided below and on disk (in the Krueger drawer) accomplishes the same thing as the assembler fragment above, but is written in C. The structure definition for the fake seglist is:

```
struct FAKE_SegList {
   long space;
   long length;
   BPTR nextseg;
      short jmp;
      void (*func)();
};
```

The first three fields correspond to the three fields specified in the AutoDocs. The jmp field is initialized to the hexidecimal value for the JMP instruction. The function pointer, func, is loaded with the address of the function process_starter(). These two fields form the first code hunk for the new process that will be created by CreateProc(). By using this method, you avoid using any assembly language stub routines.

After the fake seglist is created, it is passed to CreateProc(), with the process name, priority, and stack size. Remember, the seglist pointer must be converted to a BPTR. CreateProc() will create a new process that will begin executing at the JMP instruction of the fake seglist, and then jump to process_starter(). This function waits for the start-up message, performs the necessary initialization, calls the designated user function, and then replies to the start-up message, signifying that the child process has ended.

The child waits for the start-up message by calling WaitPort() with message port from its process structure. When WaitPort() returns, the start-up message is ready to be fetched. You accomplish this with GetMsg(). The message can be of any size, containing any information necessary to run the new process. The only restriction is that the first item in the message be a struct Message. This example uses the start-up message to pass two pieces of additional information to the child process—the global data pointer and a pointer to the function that the user wishes to execute. The structure used to hold this information is struct ProcMsg and is defined in process.h. The start-up message is created by allocating memory and filling in the desired fields. It is passed to the child process via PutMsg().

Now that the parent process has sent the start-up message and the child has received it, both processes can run simultaneously. Eventually, the parent process must wait for the child process to finish. In the example, wait_process() accomplishes this by waiting on the reply port of the start-up message. This is how the child process will signal its completion. When the reply is received, the parent knows the child has finished, and it is safe to free the reply port and memory associated with the start-up message. Note that the return code is passed from the child processes to the parent in the return_code field of the start-up message. This value is extracted before the start-up message is freed.

LOOK IT UP

In the Krueger drawer, the example consists of three files—main.c, process.c, and process.h. The first, main.c, is just a driver program that calls the routines in process.c. It contains three functions—main(), process1(), and process2(). By using the routines in process.c, the main() function creates two child processes that run simultaneously, and then waits for them to finish. The functions process1() and process2() contain the code that the child processes will execute. Note that both process1() and process2() call Execute(), demonstrating that it is possible to combine the creation of synchronous and asynchronous processes.

The file process.c contains the routines necessary to create new processes and wait for their termination. They may be used "as is" or modified as desired. The file process.h contains the necessary prototypes and structure definitions for process.c.

To use these routines, include the process.h header file and call the functions as follows:

```
msg = start_process(int (*func)(), long priority, long stacksize);
ret = wait_process(struct ProcMsg *msg);
```

where

msg	is a struct ProcMsg * defined in process.h.
func	is the function that will be the entry point of the new process.
priority	is the priority of the new process.
stacksize	is the stacksize of the new process.
ret	is the return from the new process.

Remember to be careful about which functions you use to create new processes. All functions must be re-entrant and ►

must not call exit(), as exit() will free all the memory allocated by the main program while the main program continues running.

You can customize the files process.c and process.h. Areas of interest are indicated by comments containing the string "user." These areas indicate where you can add code to pass additional information to the child process.

The above example uses code from the current program as the code for the new process. If you wish to create a new process that executes code from a separate program, then use LoadSeg() to obtain a seglist.

LoadSeg() takes one parameter, the name of the program to load, and returns the seglist. If it is unable to load the program, LoadSeg() returns NULL. You can then use the seglist LoadSeg() provides to call CreateProc(). After this call, everything is the same as in the previous example, with one exception: When the child process finishes executing, you must call UnloadSeg() to free the memory being used by the child process.

This is the technique that the SAS/C routines forkl() and forkv() use to create child processes. The fork() routines call LoadSeg() and CreateProc(), while wait() is responsible for waiting for the child process to finish and calling UnloadSeg(). Remember that LoadSeg() will not load a command from the resident list, and, therefore, you cannot use the fork routines to execute any resident command.

COMMUNICATION BETWEEN PROCESSES

The example described above uses the start-up message to communicate all the information to the child process. However, messages may be sent at any time+ and to any process. All you need is a message port that is known by both processes. For the example above, in which the parent creates the child process, the parent could create another message port and pass it to the child via the start-up message. In the case of two independent processes, communication is accomplished through a global message port. Communication with the AREXX process is an example of this type of message passing.

Signals may also be used to communicate between processes. This method is generally used to signal the termination of a process. For example, assuming that the child process was set up to handle a CTRL-C signal, the parent could execute the following to signal the child to break:

```
SetSignal(child_process, SIGBREAKF_CTRL_C);
```

The example above is not set up to receive signals. If a child process created by the example code receives such a signal while it is performing level-two I/O, the child process will attempt to call the break handler routine. By default, break handler will call exit(), which will free all allocated memory (including the parent's memory) before terminating. To avoid this situation, the default break handler is set to NULL before any child processes are created. In SAS/C, this is accomplished with the line:

```
_ONBREAK = NULL;
```

SHARING MEMORY

Because the Amiga has a single address space, all memory is available to all processes. This makes sharing memory between processes very easy, but also very dangerous. For example, imagine the case in which one process is traversing a linked list, while another is modifying it. The process traversing the list could start accessing freed memory. To avoid such situations, all references to memory that can be modified must be surrounded by Forbid() and Permit() calls. In the example above, register A4 is passed to the child process, allowing the child and parent processes to use the same global data. Therefore, all areas of code that access global data that is not read-only must be surrounded by Forbid() and Permit() calls. It is very important to be cautious of the library routines you use as well.

Avoid all library routines that are not re-entrant. The SAS/C memory allocation and file I/O functions are two examples of library routines to be avoided. Child processes should use AmigaDOS system calls for memory allocation and file I/O. The memory-allocation routines may be used if extreme care is taken to surround the calls with Forbid() and Permit(). Such stubs as the following may be used, but they must be used by all processes sharing the global data, *including* the parent process.

```
void *my_malloc(length)
int length;
{
   void *ret;
   Forbid();
   ret = malloc(length);
   Permit();
   }
```

This technique will not work with the file I/O routines, as they call the AmigaDOS I/O routines that break Forbid() calls.

THE SAMPLE PROGRAM

The sample program has been provided in Workbench environment for the SAS/C 5.10 compiler. You may have to make minor modifications for the program to run under DICE or Manx C. Two locations where changes for Manx C are required are denoted by comments.

Open the drawer containing the example. You should see several icons. Double click on the BUILD icon to compile and link the program. Double click on the TEST icon to run the program. Three windows will appear. The first is the standard window that opens as standard output for SAS/C programs. Two more windows will open. The output that goes to these to windows comes from two child processes running simultaneously. The first process calls Execute() for the DIR command, while the second calls Execute() for ECHO. Both processes call Delay() so the windows do not close before you get to see the output.

The multitasking ability of the Amiga is a valuable asset. The code provided here makes it easy to create child processes that can handle a wide variety of tasks. Because most programs wait for user input for a large percentage of the time, why not take advantage of this multitasking ability and start using those idle CPU cycles. ■

Steve Krueger is a Systems Developer for SAS Institute. He was responsible for a major part of the 5.10 release, including LSE, LC2, BLINK, and ASM. Steve is currently working on the next Amiga release. Other projects include code generators for the Apollo 3000 series and the HP700 series. Contact him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or on BIX (skrueger).*

Programming 2.0's NewMenus

More time-savers from gadtools.library.

By Paul Miller

IF YOU HAVE ever been frustrated with creating menu-strips by hand, then you will love OS 2.0's NewMenus. In addition to the new gadget classes I outlined in the August/September '91 "Graphics Handler" (p. 14), gadtools.library provides a very simple new menu format. Under it, you can lay out an entire strip, including titles, items, and subitems, with one, easy-to-understand array of structures. GadTools handles all the details of menu and item positions and sizes, linking of items, and text layout for you. (1.3 programmers do not despair. Keep reading; I have a little surprise for you later.)

THE APPETIZER

To create a menu layout definition with NewMenus, all you have to do is fill out an array of NewMenu structures. Each element of the array will define a new menu title, item, subitem, or the end of the strip. The NewMenu structure is declared in libraries/gadtools.h:

```
struct NewMenu
{
    UBYTE nm_Type;          /* See below */
    STRPTR nm_Label;        /* Menu's label */
    STRPTR nm_CommKey;      /* MenuItem Command Key Equiv */
    UWORD nm_Flags;         /* Menu or MenuItem flags */
    LONG nm_MutualExclude;  /* MenuItem MutualExclude word */
    APTR nm_UserData;       /* For your own use */
};
```

The nm_Type can be NM_TITLE, NM_ITEM, NM_SUB, or NM_END. For example, here's a simple menu definition:

```
struct NewMenu sample_menu[] = {
    {NM_TITLE, "Project",   NULL, NULL, NULL, NULL},
        {NM_ITEM, "New",        "N",  NULL, NULL, NULL},
        {NM_ITEM, NM_BARLABEL, NULL, NULL, NULL, NULL},
        {NM_ITEM, "Open...",    "O",  NULL, NULL, NULL},
        {NM_ITEM, "Close",      NULL, NULL, NULL, NULL},
        {NM_ITEM, NM_BARLABEL, NULL, NULL, NULL, NULL},
        {NM_ITEM, "Save",       "S",  NULL, NULL, NULL},
        {NM_ITEM, "Save As...","A",  NULL, NULL, NULL},
        {NM_ITEM, NM_BARLABEL, NULL, NULL, NULL, NULL},
        {NM_ITEM, "Quit",       "Q",  NULL, NULL, NULL},

    {NM_TITLE, "Edit",      NULL, NULL, NULL, NULL},
        {NM_ITEM, "Undo",       "U",  NULL, NULL, NULL},
        {NM_ITEM, NM_BARLABEL, NULL, NULL, NULL, NULL},
        {NM_ITEM, "Cut",        "X",  NULL, NULL, NULL},
        {NM_ITEM, "Copy",       "C",  NULL, NULL, NULL},
        {NM_ITEM, "Paste",      "V",  NULL, NULL, NULL},

    {NM_END, NULL, NULL, NULL, NULL, NULL}
};
```

As you can see, it is relatively easy to determine what the menu strip will look like. Note that you no longer have to deal with IntuiText structures—gadtools.library creates them for you. GadTools will automatically generate a separator-bar item for you if you specify the nm_Label field as NM_BARLABEL. You should also note that menus and items are now enabled by default. If you want to disable a menu or item, use NM_MENUDISABLED or NM_ITEMDISABLED, respectively, in the nm_Flags field. The CHECKIT, MENUTOGGLE, and CHECKED flags are still around, of course, as well as mutual exclusion and the item highlighting flags. An additional field, nm_UserData, lets you attach some of your own information to each menu item for the program act upon as you see fit when the item is selected. This is a good place to stick a function pointer for that item. When the item is selected, you could call the function attached to the item, without needing to parse the returned menu ID to figure out which item was selected.

You might have noticed by now that there does not seem to be a way to attach IMAGES to items anymore. You're partially right—there only *seems* to be no way of doing it. If you want to use an Image as an item or subitem, you need to use IM_ITEM or IM_SUB as the nm_Type specifier. In these cases, nm_Label points to an Image structure.

THE MAIN COURSE

Gadtools.library provides a set of functions that assist in allocating and deallocating standard 1.3 menu-strips, based on the NewMenu layout. To create a menu that is ready for attaching to a window all you need to do is pass your NewMenu array pointer to the GadTools function CreateMenus(), which performs all the Menu, MenuItem, and IntuiText structure allocations and linking. For example:

```
menu = CreateMenus(newmenu, tag1, ...);
```

CreateMenus() returns a pointer to a standard, dynamically allocated 1.3 menu-strip that you can parse and play with just like in the old days. You will need to do a little extra work, however, if you use the UserData field provided by NewMenus. Because there is no space reserved in the 1.3 Menu and MenuItem structures, when allocating its memory CreateMenus() tacks on a few bytes to make room for you. You must use a couple of special macros to get access to your data. To access the UserData field of a Menu, use the GTMENU_USERDATA() macro, and pass it a pointer to the de- ►

sired Menu structure. For an item, use GTMENUITEM_USERDATA(), and pass a MenuItem pointer.

Now that you have your allocated menu-strip, you must send it to one more initialization function: LayoutMenus() handles all of the positional and size calculations, based on the specified Menus, MenuItems, display VisualInfo data, and font.

```
success = LayoutMenus(menu, visual_info, tag1, ...);
```

Consequently, gadtools.library must know everything about the font used in drawing the menu. This font information is passed to LayoutMenus() through the use of tags.

If you are unfamiliar with 2.0's tag facility, here is a quick review. (For a complete discussion, see "Digging Deep in the OS," p. 8.) Tags are an extensible method of adding features and parameters to many new Intuition objects, and they form the basis for specifying GadTools gadget options. Tags consist of a tag type and data pair, and are specified in one of two ways. First, tags can be passed to most functions through a variable argument system, where one or more tag pairs is passed on the stack, followed by an end-tag (TAG_END or TAG_DONE). You can also send tags as an array of TagItems (defined in utility/tagitem.h), followed by an end TagItem.

CreateMenus() and LayoutMenus() both accept tags as variable arguments. These actually call CreateMenusA() and LayoutMenusA(), respectively, which accept an array of TagItem structures.

Currently only a few tags are supported by NewMenus. The desired font is specified with the GTMN_TextAttr tag and is accepted by LayoutMenus(), followed by a pointer to a filled-out TextAttr structure describing the font. LayoutMenus() returns TRUE if the font could be opened successfully. Otherwise, it returns NULL.

You can specify the color that menu-item text is rendered in by sending the GTMN_FrontPen tag to CreateMenus(), followed by the desired pen color.

You can find out error information from CreateMenus() by sending the GTMN_SecondaryError tag, followed by a pointer to a ULONG, initialized previously to NULL. If an error occurs during menu-strip creation, one of the following conditions are set in the variable:

GTMENU_INVALID: The NewMenu structure describes an illegal menu. CreateMenus() will return NULL.

GTMENU_TRIMMED: The NewMenu structure has too many menus, items, or subitems, and the resulting menu will be trimmed down.

GTMENU_NOMEM: There was not enough memory for CreateMenus() to allocate the entire menu. It returns NULL.

If you pass the GTMN_FullMenu tag to CreateMenus(), followed by data set to boolean TRUE, CreateMenus() is forced to build menus based on complete NewMenu arrays. If a menu fragment is discovered, CreateMenus() returns NULL.

Once you create your menu-strip with the simple gadtools.library NewMenu system, the scenario is just like in 1.3. You should have a complete, 1.3-compliant menu-strip (except for the extra UserData information), ready for use.

Intuition has a new function, ResetMenuStrip() that works exactly like SetMenuStrip() except its faster, and is useful only if the menu-strip layout has not changed. Here is the AutoDoc-condoned sequence of events for handling your menus:

1. Call OpenWindow().
2. Call SetMenuStrip().
3. Perform zero or more iterations of the following steps:
   Call ClearMenuStrip().
   Change CHECKED or ITEMENABLED flags.
   Call ResetMenuStrip().
4. Call ClearMenuStrip().
5. Call CloseWindow().

"When you are finished, you can dispose of an entire menu-strip by passing the menu pointer to the FreeMenus() function."

GadTools provides one more function for dealing with NewMenus. When you are finished, you can dispose of the entire strip by passing the menu pointer (originally given to you by CreateMenus()) to the FreeMenus() function.

AND FOR DESSERT...

What can you do if you don't currently have access to 2.0, but want the simplicity of creating menus using the NewMenus facility? Fear not, for in the Miller drawer of the accompanying disk I have included a set of reverse-engineered functions that (for the most part) act like the gadtools.library NewMenu equivalents. Link newmenus.o with your standard 1.3 code, and you will have the ease-of-use of 2.0 NewMenus too! This module is in SAS/C format, so doublecheck the source before you recompile it for a different environment. Note that newmenus.o does not incorporate *all* of the features of version 37 NewMenus, but my test menu-strip (used in the example code) works the same using both GadTools' and my own functions. See the source for details on what has been left out or does not work quite as expected. In case you do not have access to 2.0 includes, I have added a header file (newmenus.h) that defines all of the required structures and tags used by the NewMenu functions. Include this in place of libraries/gadtools.h and you are all set!

NewMenus provides a simple, extensible new format that makes it much easier to create, edit, and understand complete menu-strips. Coupled with powerful functions that handle the gory details of consistent dynamic menu allocation, layout, and deallocation, gadtools.library's NewMenus brings sanity back to managing your menu! ■

Paul Miller has been a developer since 1985 and is currently majoring in Computer Science at Virginia Tech. Contact him c/o The AmigaWorld Tech Journal, 80 Elm St., *Peterborough, NH 03458, or via Internet (pmiller@vttcf.cc.vt.edu).*

NEW 1991 RELEASES!

SAVE $5! EACH

3 Exciting New Videos Packed With Inside Info And Hot Tips!

HURRY WHILE SUPPLIES LAST!

1-800-343-0728

CALL TOLL FREE or mail this coupon.

VIDEO TOASTER

The Video Toaster™ from NewTek is hailed as the world's first video computer card enabling broadcast-quality production on desktop! TheVIDEO TOASTER™ videotape is indispensable for Amiga owners considering the purchase of a Toaster™ or those curious about all the excitement over this "revolutionary breakthrough in technology."

VIDEO TOASTER™ provides in-depth, comprehensive information on the Toaster's™ wide array of features and amazing capabilities. Topics covered include installing the Toaster™ in the Amiga 2000; adding and testing other essential equipment; selecting source material; and manipulation of the many digital video effects, including flips, tumbles, mirrors, spins, splits and titles. This video also illustrates how to generate and then superimpose letters over pictures, how to produce three-dimensional animations and how to paint on video images.

See for yourself what the excitement is all about!

HOT ROD!

HOT ROD YOUR AMIGA provides authoritative advice on how to achieve maximum power with your machine, whether you own a series 500, 2000 or 3000 Amiga.

HOT ROD YOUR AMIGA teaches you how to expand memory internally and externally. It provides valuable, in-depth information on selecting and installing hard drives, memory boards and accelerators; back-up software and utilities; RAM and drive space differences; and other "hot-rodding" tips. It also covers high-end peripherals such as DCTV™ and the revolutionary Video Toaster™. Don't wait to soup up your Amiga!

PRIMER

THE AMIGA PRIMER video provides step-by-step instructions covering the many features of the Amiga. Whether you're a new owner or an experienced user, this easy-to-follow video will prove invaluable. Packed with almost 90 minutes of detailed information, THE AMIGA PRIMER teaches you in an entertaining format with vibrant graphics and upbeat music.

Gain the full benefits that the Amiga has to offer on all Amiga models, System 2.0 and AmigaVision®. It also covers the Amiga workbench, the CLI, peripherals and utilities. There's no easier way to master your Amiga!

YES! I am eager to become an expert! Please send me the following videos:

❑ Video Toaster™	~~$24.95~~	*$19.95*
❑ Hot Rod Your Amiga	~~$24.95~~	*$19.95*
❑ The Amiga Primer	~~$29.95~~	*$24.95*
❑ Animation Video, Vol. One	~~$19.95~~	*$14.95*
❑ Desktop Video, Vol. One	~~$29.95~~	*$24.95*
❑ Amiga Graphics, Vol. One	~~$29.95~~	*$24.95*
❑ The Musical Amiga	~~$29.95~~	*$24.95*
❑ Animation Video, Vol. Two	~~$24.95~~	*$19.95*

❑ Check/Money Order ❑ MasterCard ❑ VISA ❑ AmEx
❑ Discover

Make checks payable to TechMedia Video.

Please include $2.95 shipping & handling for one video, $5.00 for two or more.

Canadian orders add 7% GST (GST reg. #126038405) **Total Amt. $** _____

Acct. # _____ Exp. Date _____

Signature _____

Name _____

Address _____

City/State/Zip _____

TECHMEDIA VIDEO

TJ31291

PO Box 802, 80 Elm Street, Peterborough, NH 03458 603-924-0100

An IDG Communications Company

Programming Serial.Device

By Robert Wittner

PROGRAMMERS WHO NEED better control over serial communications than SER: allows or need a reliable way to perform serial input and output (I/O) can use the Amiga's serial.device directly. As with most other Amiga devices, the serial device uses Exec's basic I/O functions and builds upon Exec's data structures. Knowing how to program the serial device will give you control over all serial parameters, let you perform extensive error-checking and asynchronous I/O, plus allow you to handle multiport serial boards effectively. If you are not comfortable with the basics of Exec I/O, consult Chapter 19 of the *Amiga ROM Kernel Manual: Libraries & Devices* before you begin.

INITIAL STEPS

There are a few steps that must be completed before any serial communication can take place. First, you must create an Exec message port to receive notifications from the serial device. To do so, you use the CreatePort() function, which accepts two arguments and returns a pointer to a newly created message port (or NULL if the call fails). The first argument is a pointer to the port's name or NULL if the port is to be anonymous. The ports that are used in conjunction with the serial device should be anonymous, because no other process will be looking for them and assigning names to the ports only causes needless system overhead. The second argument is the port's priority, which applies only to named ports and will not be used in the following examples. CreatePort()'s complement is DeletePort(), which accepts one argument: a pointer to a port to be removed. You create a port as follows:

```
struct MsgPort *MyPort;

if (!(MyPort = (struct MsgPort *)CreatePort(NULL, 0)))
  {
    /* CreatePort failed, place an error handler here! */
  }
```

Next, you create and initialize the request structure that you will use to issue commands and pass parameters to and from the serial device. The IOExtSer structure serves the purpose. An extension of the Exec IOStdReq structure, the IOExtSer structure is defined as:

```
struct IOExtSer
  {
    struct IOStdReq   IOSer;
    ULONG         io_CtlChar;
    ULONG         io_RBufLen;
    ULONG         io_ExtFlags;
    ULONG         io_Baud;
    ULONG         io_BrkTime;
    struct IOTArray   io_TermArray;
    UBYTE         io_ReadLen;
    UBYTE         io_WriteLen;
    UBYTE         io_StopBits;
    UBYTE         io_SerFlags;
    UWORD         io_Status;
  };
```

The structure members will be defined as they become relevant. To initialize this structure, use CreateExtIO(), which accepts two arguments and returns a pointer to an initialized extended I/O request block. This I/O request block is the IOExtSer structure in the example. The first argument in CreateExtIO() is a pointer to an initialized message port; the second is an integer indicating the size (in bytes) of the extended I/O request block. The complement to the CreateExtIO() function is DeleteExtIO(). This function accepts a pointer to a previously created IOExtSer and disposes of it. The following C code fragment creates and initializes an IOExtSer structure:

```
struct IOExtSer *MyIOExtSer;

if (!(MyIOExtSer = (struct IOExtSer *)CreateExtIO(MyPort,sizeof(struct
IOExtSer))))
  {
    /* CreateExtIO failed, place an error handler here! */
  }
```

The next function you need is OpenDevice(), which accepts four arguments and returns a boolean value indicating success or failure. The first argument is a pointer to the name of the device to be opened. The second is the unit number of the device to which you are requesting access. The third is a pointer to your initialized IOExtSer structure created with the call to CreateExtIO(). The fourth is a bitmask (flags) and is not used with the standard serial.device. However, some devices that are serial.device-compatible, such as drivers for multiport serial boards, may use this argument. Check the documentation that comes with these peripherals for details.

The complement to OpenDevice(), the CloseDevice() function, accepts one argument: the pointer to the IOExtSer structure you used as the third argument in the OpenDevice() call.

The structure member MyIOExtSer->io_SerFlags contains a bit labeled SERB_SHARED. If this bit is set when you call OpenDevice(), it allows other programs to access the same

Go a level deeper and gain more control

over your programs.

port that your program is using. This could be exactly what you want, or it could spell disaster. If you want to ensure that your program has exclusive use of a serial port, you should make sure that this bit is not set. To do this in our example, you can perform the following assignment:

```
MyIOExtSer->io_SerFlags &= ~SERF_SHARED;
```

Note the use of the bitmask SERF_SHARED instead of the bit number SERB_SHARED.

If you plan to use the examples given below, you must include some C header files in your code. You may also want to examine the data structures and constant declarations in these files for educational purposes. The obvious file to include is devices/serial.h, which contains all the device-specific data structures and constant definitions. In its current form, devices/serial.h automatically includes exec/io.h, which contains data structures and constants for device-independent I/O (CMD_READ, CMD_WRITE, and others mentioned below). Another possibility is exec/memory.h, if you plan to use Exec's memory-allocation capabilities. Finally, most programs make use of exec/types.h, which contains definitions for such items as UBYTE, UWORD, and other common type-defined data types.

With all this in place, you can now "open" the serial device. Note that the string "serial.device" in the example below can be replaced with the driver of any serial.device-compatible device. Also, you can change the unit number to indicate which port on a multiport board you want to use. Keeping with the previous examples, the following code allocates the built-in serial port for use by your process:

```
#define DEV_NAME    "serial.device"
#define UNIT_NUM    0
if (!(OpenDevice(DEV_NAME, UNIT_NUM, MyIOExtSer, 0)))
  {
    /* The open failed. Reasons:
       1. The named device and unit do not exist
       2. The device is being used exclusively by another program
       3. We couldn't secure exclusive use of this port
    */
  }
```

If this last step is completed successfully, the requested serial port is available for use. If you encounter an error, you must remember to dispose of what you allocated by using the functions CloseDevice(), DeleteExtIO(), and DeletePort() in the order listed here. You must perform a CloseDevice() on your allocated port, especially if you were granted exclusive access. If you fail to relinquish control of a port upon program termination, no other programs will be able to use that port until the Amiga is reset.

DATA TRANSFER

At this point, you can begin transmitting and receiving data. You have a choice of two methods. The first is synchronous I/O, in which you issue a command to the device and wait for its completion. The DoIO() function issues the command to the device and then waits for the command to complete before returning control to your process.

You have to fill in some of the IOExtSer structure members with data, however, before calling DoIO(). For this example, the IOSer.io_Data field must be filled in with a pointer to the data to be sent. For receiving data, this would point to a buffer area for storage of incoming data. Next, the IOSer.io_Length field must be assigned a number indicating how many characters are to be written (or read). Following that, the IOSer.io_Command field must contain the command to be executed, as defined in devices/serial.h. Certain commands, such as SDCMD_SETPARAMS (described below), may require that the IOSer.io_Flags field contain valid data.

The following code writes a string to the serial port synchronously:

```
MyIOExtSer->IOSer.io_Data = (APTR)"Only Amiga makes it possible!";
MyIOExtSer->IOSer.io_Length = 29;
MyIOExtSer->IOSer.io_Command = CMD_WRITE;
DoIO(MyIOExtSer);
```

Upon return, MyIOExtSer->IOSer.io_Actual will contain the actual number of characters written to (or read from) the device. Additionally, MyIOExtSer->IOSer.io_Error will contain the error code if an error occurred during the execution of the most recent command.

This first approach has one critical drawback. The function DoIO() does not return control to the calling process until the I/O is complete or an error occurs. If you issue a read command to the device and no data ever arrives at the port, your process is effectively locked out and will never regain control of the CPU. This inherent problem eliminates DoIO() as an effective means of I/O for a great number of applications, including terminal programs and bulletin board systems.

The solution is the second method of data transfer: asynchronous I/O. Asynchronous I/O involves the discrete functions normally called by DoIO()—namely SendIO(), CheckIO(), and WaitIO().

First, you use SendIO() to issue the command to the device. SendIO() is similar to DoIO() (it takes the same parameters) but does not wait for the I/O operation to complete. You can ►

then use CheckIO(), which is also similar to DoIO() but returns a value indicating the status of the requested command (sent by SendIO()). Finally, WaitIO() lets you wait for completion (if your request is not already complete), reply to the "completion" message at the port, and perform some cleanup operations.

You must be careful not to write code that calls CheckIO() in a loop to determine whether an I/O request has completed. This technique is known as "busy-waiting" and degrades performance across the entire system by stealing CPU time away from other running processes. Instead, you should use the Exec Wait() function, which wakes up the task when the request has been completed. Here's an example of a system-friendly, asynchronous read:

```
MyIOExtSer->IOSer.io_Data = (APTR)MyBuffer;
MyIOExtSer->IOSer.io_Length = 1;
MyIOExtSer->IOSer.io_Command = CMD_READ;
SendIO(MyIOExtSer);

/* Other processing can occur here... */

/* Now we wait for the request to complete */
Wait(1 << MyPort->mp_SigBit);

/* Now get rid of the reply and clean up */
WaitIO(MyIOExtSer);
```

"The ExecWait() function wakes up a task when an I/O request is completed."

MyIOExtSer->IOSer.io_Actual contains the number of characters read upon completion, MyIOExtSer->IOSer.io_Error contains any error codes, and MyBuffer contains the character that was read.

Most real-world programs have several ports that receive messages on a regular basis. To wait for any port to receive a message, you logically-OR the signal bits together from all of the involved ports and Wait() for any one (or more) of the aggregate signals to become active. Here's an example with an IDCMP message port attached to an Intuition window:

```
MyActiveSignals = Wait((1 << MyPort->mp_SigBit) | (1 << MyWindow-
>UserPort->mp_SigBit));

if (MyActiveSignals & (1 << MyPort->mp_SigBit))
   {
      /* Our serial request has completed */
   }

if (MyActiveSignals & (1 << MyWindow->UserPort->mp_SigBit))
   {
      /* There may be an IntuiMessage for us */
   }
```

It can be frustrating to attempt to read one character at a time using this method. Depending on the speed of your machine and efficiency of your code (compiler), reading one character at a time limits your throughput to about 1200–2400 baud.

To have your program keep up with 2400+ baud throughput, you must adopt a different input technique, one that involves a new command, called SDCMD_QUERY. The command should be issued synchronously (with DoIO()), because it always returns immediately. Upon return, MyIOExtSer->IOSer.io_Actual contains the number of characters waiting in the buffer, and MyIOExtSer->io_Status contains several bits (defined in devices/serial.h) that indicate the current status of your allocated port. The method's basic steps are:

1. Set up a CMD_READ request for one character and issue the request.
2. When the request completes, one or more characters have been received; the first character is already in MyBuffer.
3. At this point, check MyIOExtSer->IOSer.io_Error to see if a break has been sent or if an error has occurred.
4. Next, issue a SDCMD_QUERY request. The request should be issued synchronously.
5. MyIOExtSer->IOSer.io_Status will contain status bits indicating the current state of the serial device. This could be checked for carrier loss, ring indication, and so on.
6. MyIOExtSer->IOSer.io_Actual will contain the number of unread characters in the buffer. Now, issue a CMD_READ request for the MyIOExtSer->IOSer.io_Actual number of characters. This request should also be issued synchronously, because the characters are waiting in the read buffer.
7. Process these characters as the application dictates.
8. Go to step 1.

Here is the sample code:

```
/* Note: Assumes OpenDevice() call has
 * completed successfully.
* ProcessInput() is a dummy function which handles input */

char *MyBuffer

UBYTE Done = 0;

/* Allocate a buffer of the same size as the serial device's
 * internal buffer plus one just in case one extra character
 * arrives after we tuck away the initial character */
MyBuffer = (char *)AllocMem(MyIOExtSer->io_RBufLen + 1, 0L);
if (!MyBuffer)
   {
      /* Couldn't allocate memory, handle the error! */
   }

while (!Done)
   {
      MyIOExtSer->IOSer.io_Data = (APTR)MyBuffer;
      MyIOExtSer->IOSer.io_Length = 1;
      MyIOExtSer->IOSer.io_Command = CMD_READ;
      SendIO(MyIOExtSer);

      /* Now we wait for the request to complete */
      Wait(1 << MyPort->mp_SigBit);

      /* The character is in MyBuffer */

      MyIOExtSer->IOSer.io_Command = SDCMD_QUERY;
      DoIO(MyIOExtSer);

      if (MyIOExtSer->IOSer.io_Actual)
         {
            /* There are characters in the buffer */
            /* Store them right after the character we just received */
```

```
            MyIOExtSer->IOSer.io_Data = (APTR)(MyBuffer + 1);
            MyIOExtSer->IOSer.io_Length = MyIOExtSer->IOSer.io_
               Actual;
            MyIOExtSer->IOSer.io_Command = CMD_READ;
            DoIO(MyIOExtSer);

            Done = ProcessInput(MyBuffer,MyIOExtSer->IOSer.io_
               Actual + 1);
         }
      }
   /* Return our buffer memory when finished */
   FreeMem(MyBuffer, MyIOExtSer->io_RBufLen);
```

CHANGING PARAMETERS

With a few exceptions, you can change most of the serial port's parameters at any time with the SDCMD_SETPARAMS command. This command modifies the operation of your port according to the values you place in the following variables before issuing the command:

```
MyIOExtSer ->io_CtlChar      Control characters for XON/XOFF
MyIOExtSer ->io_RBufLen      Size of the device's input buffer
*** Note: The buffer is re-allocated when you change this value, so all
data in the buffer is LOST!
MyIOExtSer ->io_ExtFlags     Controls MARK/SPACE parity
MyIOExtSer ->io_Baud         Baud rate for reads and writes
MyIOExtSer ->io_BrkTime      Break duration in microseconds
MyIOExtSer ->io_TermArray    Array of termination characters — See
the ROM Kernel manual about this and EOFMODE.
MyIOExtSer ->io_ReadLen      Bits per character (read)
MyIOExtSer ->io_WriteLen     Bits per character (write)
MyIOExtSer ->io_StopBits     Number of stop bits
MyIOExtSer ->io_SerFlags     Can change any flag except SERF_
SHARED, SERF_XDISABLED, and SERF_7WIRE. These must be set/
reset before the OpenDevice() call is made.
```

Here's an example:

```
/* Change to 2400 baud */
MyIOExtSer->io_Baud = 2400;
MyIOExtSer->IOSer.io_Command = SDCMD_SETPARAMS;
DoIO(MyIOExtSer);
```

FAST I/O

To obtain the highest possible throughput, you should call another library routine that was designed with speed in mind—BeginIO(). You can treat BeginIO() almost the same as SendIO() with one important exception: the IOB_QUICK bit in MyIOExtSer->IOSer.io_Flags. To exploit this difference, turn on the bit IOB_QUICK in MyIOExtSer->IOSer.io_Flags and call BeginIO(). When BeginIO() returns to you, check the status of the IOB_QUICK bit. If the bit is still set, the request has already completed, without the need for a reply message, a possible Wait(), or a task switch. This technique is demonstrated below:

```
MyIOExtSer->IOSer.io_Data = (APTR)MyBuffer;
MyIOExtSer->IOSer.io_Length = 1;
MyIOExtSer->IOSer.io_Command = CMD_READ;
MyIOExtSer->IOSer.io_Flags |= IOF_QUICK;
BeginIO(MyIOExtSer);

/* Test to see if the request has already been completed */
if (MyIOExtSer->IOSer.io_Flags & IOF_QUICK)
   {
      /* Request has completed */
      /* The character is in MyBuffer */
      /* No WaitIO() is necessary */
   }
else
   {
      /* Not Quick I/O so we wait for the request to complete */
      Wait(1 << MyPort->mp_SigBit);


      /* Now get rid of the reply and clean up */
      WaitIO(MyIOExtSer);
   }
```

SIMULTANEOUS REQUESTS

Applications such as terminal programs need to have a read request pending while being able to write to the serial device. You can accomplish this by setting up two message ports attached to the same device-request structure: Create a message port and an IOExtSer structure, then open the device as in the example given earlier. When this is completed, create a second message port and a second IOExtSer structure.

You then copy the first IOExtSer structure to the second IOExtSer, but change the mn_ReplyPort of the second structure to point to the second message port. For example:

```
struct MsgPort *MyPort1, *MyPort2;
struct IOExtSer *MyIO1, *MyIO2;

if ((!(MyPort1 = (struct MsgPort *)CreatePort(NULL, 0)))
   || (!(MyPort2 = (struct MsgPort *)CreatePort(NULL, 0))))
      {
      /* Couldn't create one or both ports */
      }

if ((!(MyIO1 = (struct IOExtSer *)
         CreateExtIO(MyPort1, sizeof(struct IOExtSer))))
   || (!(MyIO2 = (struct IOExtSer *)
      CreateExtIO(MyPort2, sizeof(struct IOExtSer)))))
   {
      /* Couldn't create one or both IOExtSer structures */
   }

CopyMem(MyIO1, MyIO2, sizeof(struct IOExtSer));

MyIO2->IOSer.io_Message.mn_ReplyPort = MyPort2;

/* We can now use MyIO1/MyIO2 independently */
```

ON DISK

The source code fragments contained in this article are available on the accompanying disk in the Wittner drawer. I have also included the C source code and executable file for a small, generic terminal program that demonstrates the techniques shown in this article. Study them and you will soon be ready for your own projects. ■

Robert Wittner, a Programmer Analyst for Rockwell International, is working on the space station Freedom program and has been programming the Amiga four years. Contact him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458.*

ON DISK

Hard Disks: How Fast Are They Really?

By Michael Sinz

500 K/second ... 34 files/second ... this drive ... that controller ... DMA ... nonDMA ... multitasking friendly ... video speed ... millisecond access times ... SCSI ... ST-506 ... AT ... IDE ... Adaptec ... OMTI ...

The amount of confusing, conflicting, and just plain wrong information about hard drives is extreme. Maybe the reason is that the Amiga used to have slow hard drives or that the Amiga now has some of the fastest hard drives in the industry. Much of it is because of a misunderstanding about what the various terms and numbers mean. Before I can explain these terms and how the numbers relate to how fast the system really is, you should understand the basic technical issues involved in what a hard disk drive does—beyond simply storing data.

As you know, data within a computer is just a series of 1s and 0s. To store this data, the computer must, in some way, be able to record the 1s and 0s so that they can be read back as the same pattern that was written. One of the most popular methods is magnetic recording. In much the same way as audio tape records and plays sounds, the computer generates a signal, or sound, records it, and then decodes it before playing it back when the information is requested. Computers have done this on magnetic tape, magnetic drums, magnetic-plated media, spinning magnetic tape (which became the floppy), and sealed magnetic-plated media. This has always been one of the most complex and fastest advancing fields of computer technology. Not much more than ten years ago, sealed-media hard-disk drives (known as Winchesters) were putting a whopping 5 to 10 million bytes on 8-inch disks. Today, small 3½-inch drives can store over 1,000 million bytes.

Disk drives—from floppies to hard disks to Winchesters—work much the same at the physical level. A disk drive contains one or more round, flat discs that are coated with a magnetic particulate substance, usually some form of metal oxide. This magnetic coating stores the signal from the computer.

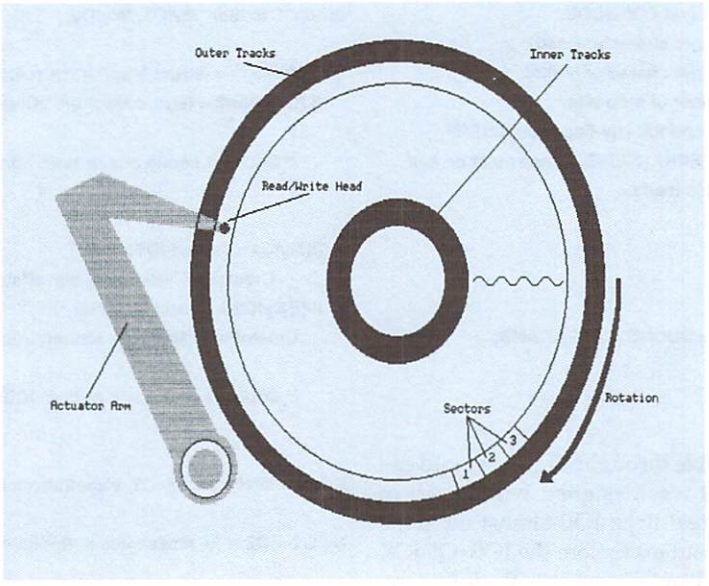


Figure 1: The basic arrangement of a hard disk.

To read and write this data, a magnetic pickup and recording module, known as the read/write head passes over the sections of the disk. Disks are divided into parts known as *tracks*. Tracks, while sounding and acting like those of an audio record, are actually fully contained loops much like those of a race track. Each track has a number of data units—called *blocks* or *sectors*—stored in a specific order. The order may not be as simple as 1, 2, 3; however, the drive (or the controller) knows and understands it. The disk in the drive rotates under the read/write head, allowing the data to pass under the read/write head for interpretation. The speed of the rotation depends on the drive and, sometimes, even on the position of the read/write head.

To pick up requested data, the read/write head must be positioned over the appropriate track. This is accomplished by an actuator arm or head stepping motor assembly. Each of the several methods of implementing head-positioning hardware has trade-offs. Some are faster at moving the head, some are more accurate, and some are much cheaper. Once the read/write head is on the right track, it has to wait for the correct data to come around for it to read. Because a specific block exists in a specific location on the disk, the drive must make sure that the read/write head is over that section of the disk during the read or write operation.

There are a number of minor differences in how these details are handled by the drive and controller. The general issues, however, are the same on all of the drive systems, be they smart SCSI drives or simple ST-506 drives. (See Figure 1.) With the basics out of the way, let's explore the impact of the above factors and what they mean to disk performance.

HOW DOES THIS RELATE TO SPEED?

Drive spec-sheets often contain information that either is unimportant or insignificant. If you read between the lines,

Find out what the specs really mean and how to test your drive's performance.

however, you can find valuable information. One of the numbers that is thrown about the most is the *average seek* (or *access*) time. This is the amount of time it takes the drive to move from its current cylinder location to the next requested cylinder. (A cylinder is a group of tracks that all have the have the same number on the various discs . For example, all the track 0s make up one cylinder.) This number is ballyhooed by many drive manufactures (and users) as each tries to one-up the other. While this does not tell you the drive's speed, it gives a relatively good first guess. Current drives run in the 15 to 25 millisecond range. For example, the average seek time for a Quantum ProDrive is 19ms.

Another value found on the spec-sheets is the *data transfer* rate. While this number may seem to be very important, few drive specifications talk about the overall data transfer rate. They usually boast a value that is the electrical transfer rate of the drive to the controller interface. For example, Quantum boasts a two-megabyte-per-second data transfer rate in ASYNC SCSI. This is, however, somewhat misleading. While the drive electronics could send data to the controller board at that speed, the data does not come from the disk at that speed. In fact, the drive data transfer speed to the physical disk may be much slower; this rate is a combination of the speed of the local drive electronics and that of rotation of the disk. The fastest a drive could read a track of data would be in one revolution of the disk. Therefore, if the disk revolves at 2400 RPM, the best speed you could expect would be 40 complete tracks in one second. (Assuming no delay between reading one track and the next.) So, a 2400 RPM drive with 32 blocks/track would get 1280 blocks/second, which translates to 640 K/second. If the drive specifications give the rotation speed of the drive and the number of blocks on a track, you can calculate the absolute maximum transfer speed.

Related to the transfer speed is the *interleave* that the drive can run on. Interleave is a trick to increase the transfer rate if the drive electronics are slow—or the host computer is. The simple case of a 1:1 interleave means that the blocks are on the disk in order. That is, block 1 is followed by block 2, and so on for the whole track. This also means that a complete track can be read in one revolution of the disk. If the time between reading one block and reading the next is too long, however, the next block may have already passed by the read/write head. If this happens, the drive will have to wait until the block comes around again before it can read it. With a 2:1 interleave, the drive has the blocks arranged on the disk as every other one. (See Figure 2.) Thus the computer or controller of the drive has more time between the blocks to get ready for the next one. It does, however, reduce the transfer speed by making the drive do more revolutions to read the track. The fact that a drive/controller combination specifies that it can run at a 1:1 interleave means that the electronics are fast enough to read the entire track in one revolution.

The other main "selling point" is the *interface* that the drive uses. There are many types of interfaces available, each with its own advantage. In general, the smarter interfaces are better as they remove much of the grunt work from the software/controller end of the equation. The de facto standard on the Amiga has become SCSI (Small Computer Systems Interface), defined by ANSI. It places a large amount of intelligence on the drive and thus makes it much easier to plug-and-play; just take a SCSI-standard drive and connect it to a SCSI controller and they will work together. SCSI is also rather fast. At two megabytes-per-sec- ►

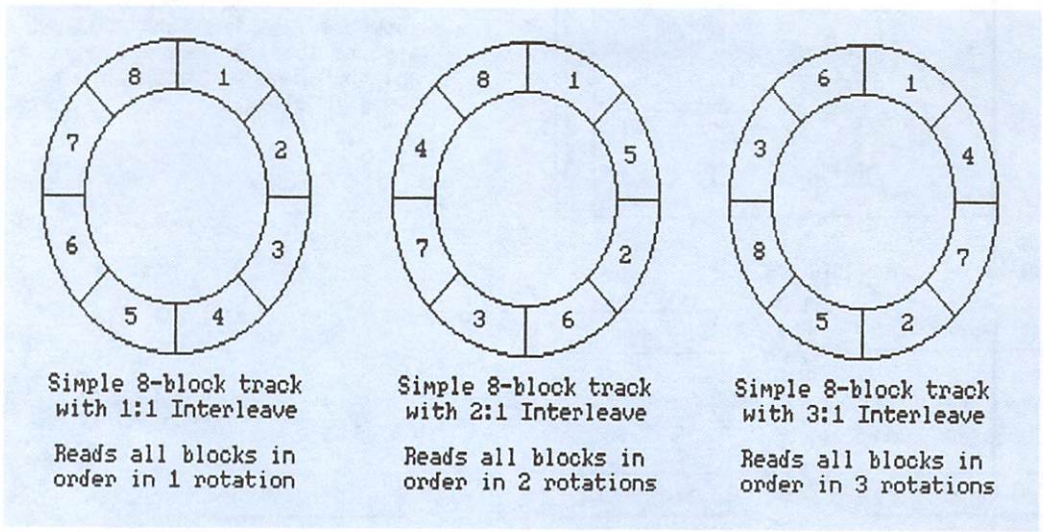


Figure 2: Three examples of interleave schemes.

ond in ASYNC transfers and over four MB/second in SYNC transfer mode, it currently is much faster than the physical limits of the drives. SCSI-2, the new specification, goes even faster. (For a complete discussion of the standard, see "Inside SCSI," p.17, Aug./Sept. '91.)

At the other end of the drive interface spectrum is ST-506, which is about as minimal as you can get. The controller software needs to know everything about the drive, including physical layout, write-precomp timings, bad-block mapping methods, and so on. Because this means a reduced amount of electronics on the drive, they were cheap and were easier to make. Today, however, the price difference has mostly disappeared. Other drive interface standards include ESDI (enhanced ST-506. with less versitility but potentially more speed than SCSI) and IDE (actually, IDE-AT and IDE-PC; reasonable intelligence and very simple to make a controller for). Other interfaces exist but are either very costly or rare.

TO DMA OR TO NOT DMA?

Actually, that question is rather simple to answer once you know exactly what the difference is. DMA stands for Direct Memory Access. As the name indicates, a DMA device has direct access to the memory system of the computer, meaning the device can actively read or write data to memory without the CPU.

Figure 3 shows a very over-simplified diagram of how a computer (the Amiga in this case) is connected. As you can see, each part of the system is connected to the main pipeline of data: the Zorro bus. (Zorro is the Amiga bus architecture.) As the small diagrams below show, a DMA transfer, once started, goes directly from the hard-disk controller board to memory—basically only one step. The CPU method, however, transfers data from the controller board to the CPU, and then out to the memory, thus taking two bus transactions.

If it were all that easy, the clear winner would be DMA. However, there are complications: When doing DMA, the controller must be told where to send the data. This takes a small amount of time, and does not need to be done in the CPU method. In addition, when the DMA is complete, an interrupt is generated to tell the system. This takes a few cycles to respond to. On the other hand, there are also some good points: First, when doing DMA, the CPU is free to use as many available bus cycles as it wants. Thus, in a multitasking system, you can continue to run other tasks while data is being transferred from the disk. The CPU method keeps the CPU very busy doing all of the work and thus leaves less CPU available for other tasks. Second, because the DMA method uses less bus bandwidth, it lets other devices that may be bandwidth-hungry work better.

Just because a controller is designed for DMA or CPU transfer, you cannot call it bad or good. Good designs, as well as bad, are possible in either format. Given two equally good implementations, however, the DMA controller will have overall better performance.

MEASURING PERFORMANCE

Measuring the performance of a disk subsystem is a rather

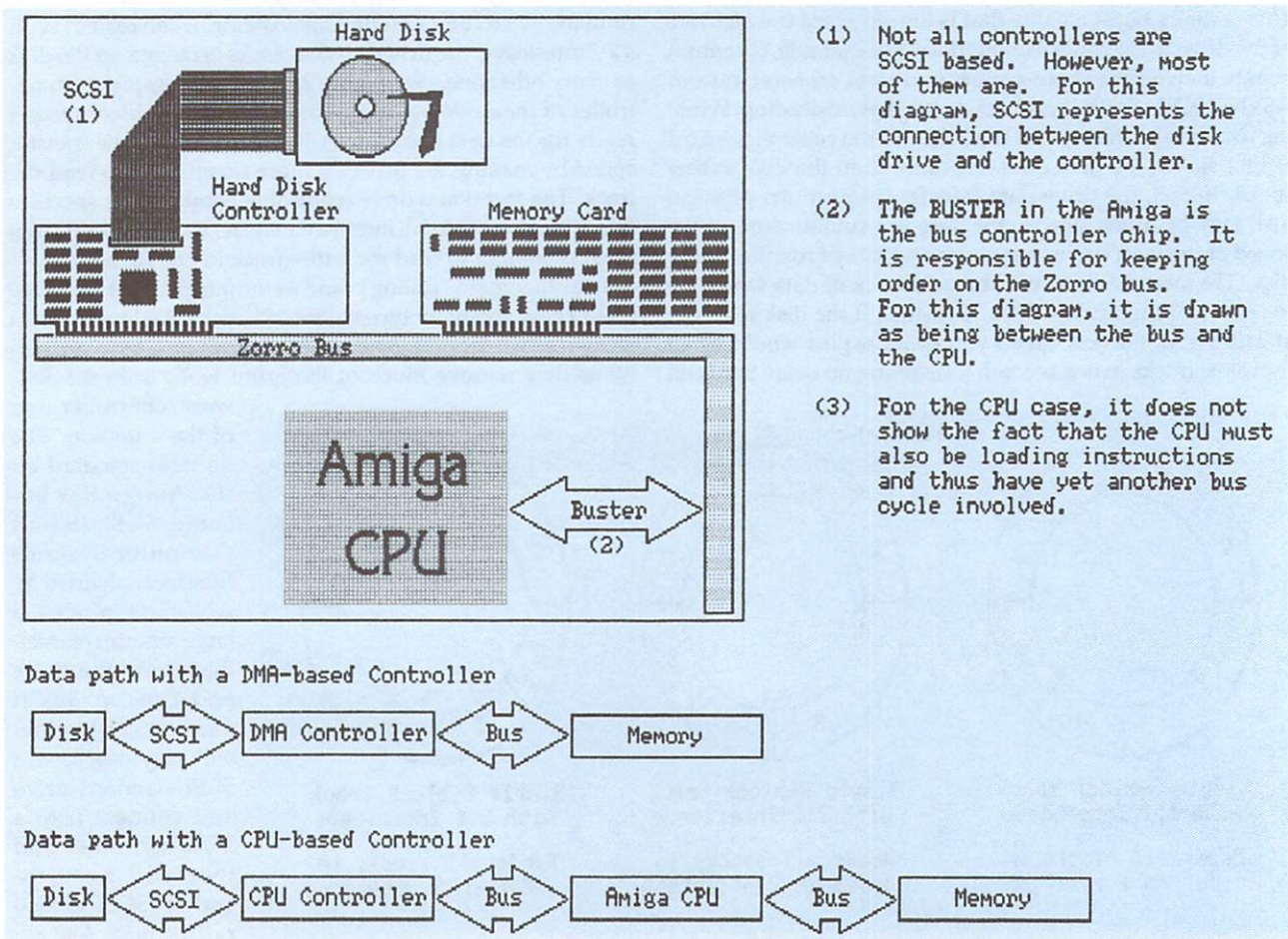


Figure 3: A simplified illustration of Amiga/ hard drive data-transfer paths.

interesting science. In addition to the physical limitations of the drive and controller, there are issues of software technology at the drive-controller, file-system, and operating-system levels. In addition, many of the standard testing issues come into play, such as accuracy of the test, accuracy of the observation, applicability of the test, and so on.

The accuracy of the test can be defined rather exactly. On the Amiga, the system has a timer that has a to $^1/_{60}$ second ($^1/_{50}$ in PAL) resolution. This comes out to roughly 0.02 seconds. Thus, any given time reading will be accurate only within +/– 0.02 seconds. To test the speed of the tests, the time must be read at the beginning and end of the test. This results in +/– 0.04 seconds of accuracy. Thus, to make the test have a +/– 1% accuracy, it would have to run for a minimum of four seconds.

The accuracy of the observation is much more difficult to quantify. The issue here is that in doing the observation, the test and thus the results are affected. The best that can be done is to try to minimize the effect of observing the test while not compromising the quality of the observations.

What the last issue—the applicability of the test—really means is how well the test (and its results) relates to the real-use performance of the drive. In many ways, this is more important than the other two issues, as without reasonable applicability the test results would be useless.

With DiskSpeed, the disk-performance test software that MKSoft Development is developing, attention has been paid to make the tests both accurate and realistic. DiskSpeed 3.1 has proven accurate and has become the standard by which Amiga hard disks and controllers are judged. With DiskSpeed 4.0 (currently under development) a whole new set of tests will be possible.

DISKSPEED: THE AMIGA STANDARD

I first developed DiskSpeed because other disk-drive performance testers were either highly inaccurate or did not relate well to real-world disk drive usage. The accuracy issues are easy to solve; however, the applicability issues took some thinking.

In DiskSpeed 3.1, the accuracy issues were solved by making the tests take a long time, ensuring that the clock's accuracy did not adversely affect the results of the test. In addition, the tests were done with as clean a software design as possible.

With DiskSpeed 4.0, I developed a new technology that can automatically size the test time to give as accurate a result as possible. It was important that this be done only in the appropriate tests, as some tests radically change their results if they are run for more iterations.

The more important, and more difficult, part of designing a set of tests is coming up with ones that will show results that apply to the real world. In that direction, none of the tests use anything other than standard AmigaDOS file I/O calls. Some people ask me to add a test that performs direct device I/O. However, no application would do direct device I/O to open, read, write, close, or delete a file. It would not only be ridiculous, but the amount of work required to write a file system is well beyond what an application developer needs to spend time on.

Now that the tests are to perform AmigaDOS I/O only, what needs to be tested? This is where you need some knowledge of the physical limitations of the disk drives and how application software works. As you already know, the Amiga's filing system is very powerful and flexible. Much of this power is from the way data is laid out on the disk. This layout, however, makes some operations a bit slower, most noticeably listing a directory. Listing a directory makes the system read many blocks of data often, from different areas on the disk, and most applications and all users run into this performance issue during everyday use. Therefore, a test that would measure the performance of the drive/controller combination when scanning a directory would provide numbers that directly relate to user experience.

In addition to scanning directories, it is important to be able to create new directory entries, find entries, and delete them. Again, these are situations that users run into every time they use an application that does anything with a disk. All together, these tests are designed to show the performance of the file system's directory structure. Note that to make these tests fair, the number of files created in the test directory is always the same. The speed of access in a directory structure changes as the number of files change, and, if this test were to auto-size itself based on the speed of the device, the results would no longer be valid.

Another test that helps show the performance of the file system and device driver is the Seek/Read test. It helps show how well a database application may run, as database operations tend to be very disk-bound and to access various locations with a large file. The Seek/Read test reads small chunks from various locations within the file. The speed with which the file system can find the correct data location within a file and then read a part of it is directly measured by this test. (Note that the DiskSpeed 3.1 Seek/Read test was rather simplistic and produced uninteresting numbers.)

The final three tests are basic file data read and write tests:

File Write/Create: Creates a new file and fills in the data. The speed of this depends on how fast the filing system can locate new empty blocks of disk space for the file.

File Write: Writes to an old file. The performance here is determined by how well the filing system deals with rewriting the data in a file that already exists. This will usually be faster than the Write/Create test.

File Read: Reads from an old file. The performance here is determined by how quickly the filing system finds the data-blocks of a file.

With DiskSpeed 3.1, each of these three tests were done with various buffer sizes, ranging from 512 bytes to 262144 bytes. DiskSpeed 4.0 adds a few twists—each test will also happen on LONGWORD-aligned buffers, WORD-aligned buffers, and BYTE-aligned buffers. Each test is then performed in fast memory and in chip memory (if you have both available).

Also new for DiskSpeed 4.0 is provision for selecting the sizes of these buffers. While the larger-size buffers are nice to play with, remember that most older applications use a 512-byte buffer only. Many newer applications are using 4096-byte buffers, as the speed improvement by just increasing the amount of data read in one I/O call is rather significant. (DiskSpeed 3.1 helped show this fact.)

In addition to the basic tests, DiskSpeed 3.1 lets you turn on DMA and CPU stress factors. To show how well the drive/controller combination worked in a video environ- ►

ment, the DMA feature increased the amount of bandwidth the video control chips were using. CPU stress was an attempt to simulate heavy workloads in the Amiga's multitasking environment.

With DiskSpeed 4.0, the CPU stress test has been removed. It turned out to produce results that did not mean much. However, to take its place is a CPU-availability value that is reported for each test. This is a rough calculation of the available CPU percentage during the test. This is a very useful number, as it will tell if there is enough CPU time available to decompress a picture while loading the next one or to handle user input during disk I/O. Observing a test always has an impact on the results. This is a known fact, and DiskSpeed is not able to get around it. In doing the CPU-availability checking, the performance of the system may change. This is due to the fact that just the act of counting the CPU time will use some CPU time and change the dynamics of the system. However, if all tests are done the same way, the relative merits of the drives under test will still be valid.

WHY...?

So, why do the numbers come out the way they do?

- Why are small transfers so much slower?

One of the major reasons is the layout of data on the disk. As Figure 2 shows, a disk's sectors can be laid out several ways. (Most hard drives have much more than eight blocks to a track.) Large transfers require the disk drive to send the data for a number of blocks. If these were blocks one to eight, the drive could read all of them in one revolution of the disk, given a 1:1 interleave. If a program asks for only one block worth of data at a time, however, the transfer of the first block can take so long that the second block will have passed by the head before the drive is ready to read it. Therefore the disk will have to rotate around until that block is available again. In the example, a read of eight blocks transfered one at a time will take seven full revolutions after the first block is processed—seven times slower than the transfer that asked for eight blocks at once. This is worst-case. Many drives today have some caching and read-ahead capabilities that help minimize this.

- Why are the results inconsistent from one test to another?

Disk performance testing is a rather complex task. Without special equipment, many things are impossible. When DiskSpeed runs, it does not know the exact location of the disk relative to the drive heads. As a result, there is a lag between the time the drive is asked to read (or write) a block and the time that block is under the read/write head. This time lag is known as rotational latency. The faster the drive spins, the shorter this period is.

- Why does the CPU test slow the drive speed?

Depending on the method used to implement the controller software, the CPU test task, which runs at –127 priority, becomes extra overhead. The difference in speed may be rather small from the CPU standpoint, but it may be just enough to fall prey to the rotational-latency problem. Consider: When no task is running, waking up a task entails simply starting it again. If another task is running at the same time, however, the old task must first be put to sleep. This work can consume just enough time to make the system miss the next block that is coming around and require the system to wait until the information cycles past again.

- Why does drive performance change as the drive gets older?

Drive performance does not really change because of a disk's age. As files are written to the disk and then later removed, however, the empty areas of the disk become scattered. When the disk is then tested, the system must seek each of the locations where part of the data is stored. This adds seek time, rotational latency, control overhead, and processor overhead as the information is handled.

- Why are writes sometimes faster than reads?

The way the drive works can have a major impact on this. If the drive has a cache, a write can be sent to the cache while the drive is still waiting for the required position to reach the read/write head. Thus, the disk can signal that the write is completed when it is not quite done. During the time the system is getting ready for the next write, the drive will hopefully send the last write to the disk.

- Which number is most important?

The answer depends on your application and how you use your machine. If you often list directories or create files, you should pay attention to the directory-manipulation tests, including files-per-second that are created, opened, scanned, and deleted. One of the numbers that is most important to me is the small-buffer performance—that is, the performance of the drive/controller on buffered reads between 512 bytes and 4096 bytes. These two sizes are much more representative of the size of the read/write buffers of most applications. Artists and animators care more about large buffer sizes, because high-speed performance for large files is a major factor in their work. However, large buffers are useful only if the file can be read as one big chunk.

- Why does the test sometimes show more than 100% available CPU?

Because the CPU availability must be measured to get a reading of the total CPU present, the measurement can be incorrect by a small amount. The measurement code tries its best to get an accurate reading; it does not always succeed. Most of the time, however, it will notice when accurate measurements are not possible and will turn off the CPU testing, because the results will be meaningless.

With the addition of the CPU-availability numbers, a much more complete picture of drive and system performance can be obtained. As multimedia becomes more important, the performance combination of high drive speed and large amounts of available CPU power will make it all possible.

With DiskSpeed 4.0, developers will be able to ensure that the designs of their hardware and software live up to the performance needs of their users. It will also give the Amiga data that proves the performance of the system for real work. Applications such as database servers, file servers, and multimedia programs require as much performance as possible in the drive subsystem. The Amiga has the performance to outshine most other platforms in this area. ■

Michael Sinz, a Systems Software Engineer at Commodore, is responsible for many parts of the OS. Write to him c/o The Amiga-World Tech Journal, *80 Elm St , Peterborough, NH 03458, or contact him on BIX (msinz).*

From p. 16

quently, all previous software with its own embedded font coded in will lose 24 bytes from the system every time it is run. There is a function, StripFont(), that removes the TextFontExtension, but of course the old software does not use it.

The moral of that story is: *Do not define instances of graphics system structures in your code.* If a function exists to create a structure, such as GfxNew() or GetColorMap(), then use that; otherwise use AllocMem(). It will make our lives at Commodore easier in the future.

YOU HAVE THE CON

We hope you like the final results of 2.0. With its new features and stability, the graphics library has been greatly improved. In fact, the entire Kickstart is now more stable than any other previous release. There are plenty of new graphics features to make your program writing easier and make your programs more attractive. Use the database, and take note of anything marked as private! ■

Spencer Shanson is one half of the duo at Commodore that is responsible for maintaining and extending the Amiga's graphics.library. He was previously employed by British software developers, including Argonaut Software (to work on Starglider 2) and Burocare (to develop Scanners and TapeStreamers for the Amiga). Contact him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or on BIX (sshanson) or Usenet (spence@commodore.com uunet!cbmvax!spence).*

From p. 20

use problems to access addresses that can be trapped on by Enforcer.

BUY NOW, SO YOU DON'T PAY LATER

At a small developer meeting last spring, we at CATS were disappointed to discover that although we had convinced the majority of the audience that they needed Enforcer, a relatively small percentage of the developers owned the equipment necessary to run it—an MMU. If you do not have an MMU, get one! The investment in an A3000, 68030 card, or 68000+ MMU card will quickly pay for itself by cutting down on your development time and allowing you to catch and find software problems with Enforcer. Enforcer and Mungwall are not just for developers and QA departments. Anyone who uses or reviews in-house software or software for purchase or contract can benefit his company by catching hidden software problems during normal usage and examination of the programs. Many people at Commodore run Enforcer all of the time, including the Vice President of CATS. (Keep that in mind if you are trying to impress him with your software!). ■

Carolyn Scheppner is Technical Manager of CATS (Commodore Applications and Technical Support) U.S. Contact her c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or on BIX (cscheppner) or Usenet ([uunet, rutgers]!cbmvax!carolyn).*

From p. 46

ginIO() finds CMD_WRITE or SDCMD_BREAK, the request is directed to the write task. The other device commands, such as CMD_RESET, are performed "immediately"—within the context of BeginIO(). Most of those other routines are surrounded by a Forbid()/Permit() pair to avoid unpredictable interactions with the two associated tasks and reentrancy problems. (Note, however, that semaphores are usually a better choice than Forbid()/Permit(), at least for long operations.)

An important lesson I learned from writing the serial driver is to perform the work whenever possible within BeginIO(), rather than shipping it to a task. For example, if a CMD_WRITE request specifies only a one-byte length, and the transmit register is empty, then the request should ideally be performed within BeginIO(). The performance increase, compared with sending the request to a task, is quite large. Likewise, if there are enough bytes in the read buffer to satisfy a CMD_READ request, then copy the bytes to the user's buffer and return immediately.

Unlike the RAM disk example, the serial drive fully implements all device routines, including Open(), Close(), Expunge(), and AbortIO(), and so represents a complete example driver. Take a close look; in this case, a few lines of code may be worth a thousand words! ■

Dan Babcock is an electrical engineering major at Pennsylvania State University and an avid assembly programmer. Contact him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or on BIX (danbabcock).*

LETTERS

Flames, suggestions, and cheers from readers.

MORE GAMES

First off, congratulations on what looks like an excellent magazine. The disk is great to have; I'm not crazy about typing pages of source code, à la *Amiga Transactor*.

As a C programmer interested in writing games, I would like to know if your magazine is planning to run some articles on programming games. I would really like something (a library, if possible) for working with Blitter objects, as the system GEL routines are simply too slow for doing serious work.

Zoltan Hunt
Beeton, Ontario

As followups to "Arcade Elements" on page 58 of the October '91 issue, several game programming and faster graphics articles are in the works. Patience will reward you, Zoltan.

A CHALLENGE

Please pass along my suggestions on to the development community.

One thing that is rather annoying to me is that there has never been a standard for joysticks for the Amiga. (I don't call the Atari standard a very good one!) It seems to me that consiering the Amiga has a two-button mouse, a game that lets you choose between mouse and joystick usage would be handicapped if you used the joystick version. Even IBM has a two-button joystick, as ancient as it is. Why not introduce a two- or three-button standard and suggest that all developers begin using it?

My second gripe is about software that does not support '020 or '030 accelerators. When are these people going to begin developing with the future in mind? I would also like to see software that detects if the user has a 24-bit graphics card and automatically enhances the software according to the user's hardware configuration. This has to be the wave of things to come.

Tony Gore
Charlotte, North Carolina

MORE RESPECT AND COVERAGE, PLEASE

I work for a major engineering firm where there is a conflict over the use of C versus FORTRAN. This reflects the significant disparagement FORTRAN has received in the Amiga community. At my firm, C was recently chosen as the "standard" programming language, meaning all new programs must be written in C. For me, it meant unequivocal justification for the purchase of a FORTRAN compiler. I prefer FORTRAN over C because:

1. Most scientific and engineering programs have been, and still are, written in FORTRAN. First, because the functions (SQRT, EXP, LOG, and so on) in C are by default double precision. Because in FORTRAN they are single precision (perfectly sufficient for most variables), FORTRAN programs run much faster than equivalent programs in C. Second, working and verified code can be readily shared among colleagues and easily used in new programs. Verification of code, required by many clients and all government agencies, is an expensive process.

2. FORTRAN compilers are much more stable in their standardization. I have read too many times that "this program was compiled and linked using C compiler/linker X and may require alteration to compile/link using Y." It's bad enough that C compiler/linkers are inconsistent on the Amiga platform. What if I want to upload a program to another mainframe, workstation, or PC? I've had very few problems doing this using FORTRAN.

At my firm, the C decision was made by the "professional computer programmers" (those who know everything about structures and operating systems and nothing about numerical analysis and nonlinear regression) without consulting those of us who use and develop 90% of the programs (that entail simple arrays and few graphics) for the benefit of clients and company profits.

What got me going on this topic (again) was the absence of FORTRAN from the list of languages on which *The AmigaWorld Tech Journal* brings insight and benefit. Please consider publishing FORTRAN articles.

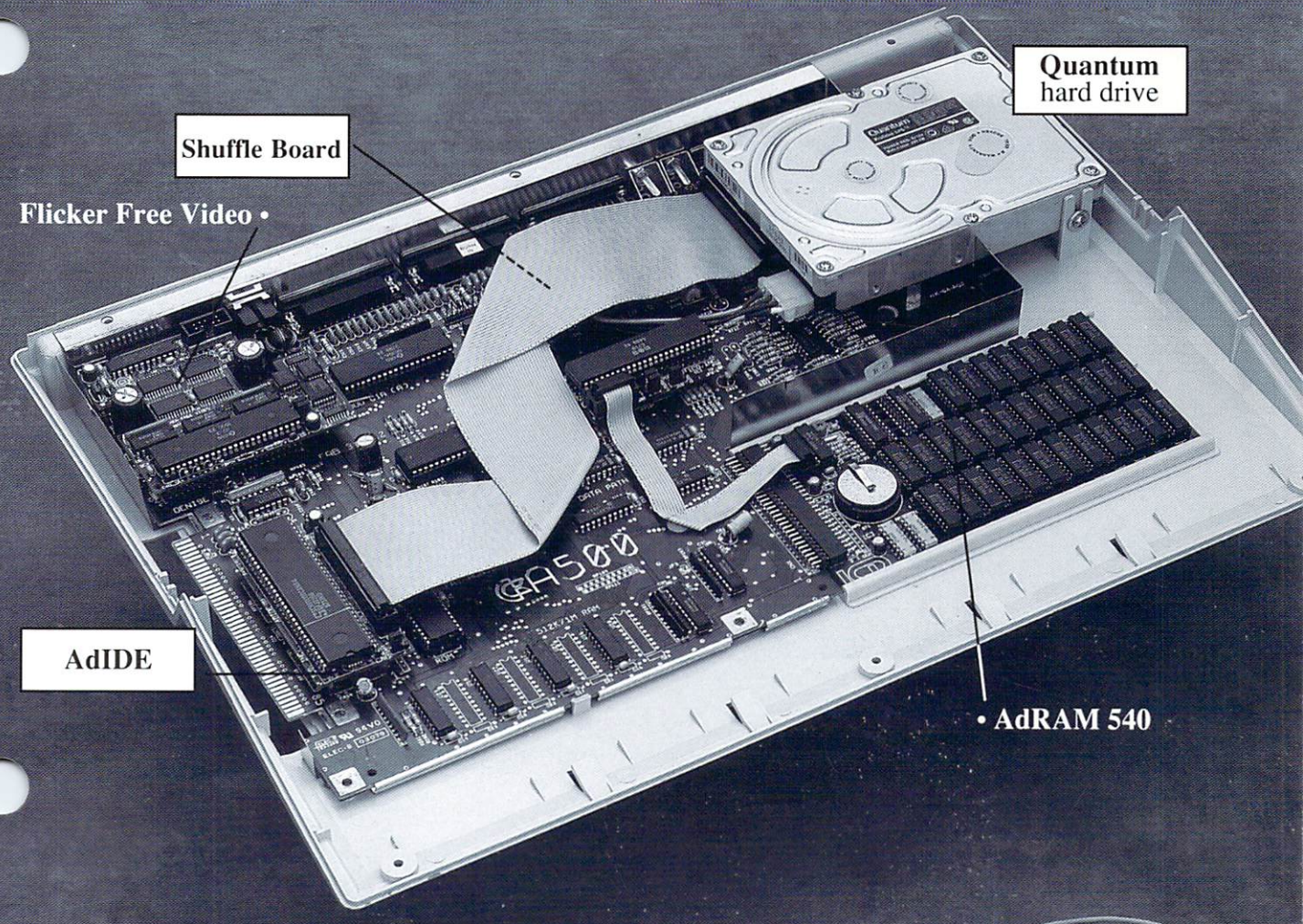
Jim Marrone
El Sobrante, California

LET US KNOW

What are your suggestions, complaints, and hints for the magazine, Amiga developers, and your fellow readers? Tell us about them by writing to Letters to the Editor, The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or posting messages in the AW.Tech Journal conference on BIX. Letters and messages may be edited for space and clarity.* ■

Prima!

A Look Inside the Ultimate A500.



ICD proudly presents **Prima**™.. the high performance, low cost hard drive for Amiga® 500 computers. Prima blends a large capacity, low power Quantum™ hard drive with the **AdIDE**™ host adapter for an unbeatable combination.

Prima replaces the internal floppy drive but includes **Shuffle Board**™ to make your external floppy drive DF0:. **Prima** features auto–booting from FastFileSystem partitions, high speed caching, auto–configuring, and A–MaxII™ support. Formatted capacities of 52 and 105 megabytes are currently available.

Prima comes complete with instructions, software, and all the hardware necessary for a simple, clean, no–solder installation. It does require an A500 with switching power supply, 1 megabyte of RAM, and an external floppy drive for setup and installation.

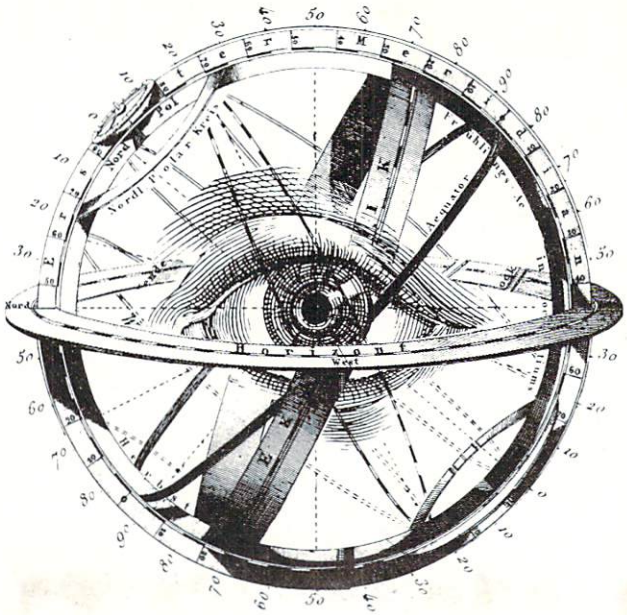
What other products would we include in the "Ultimate A500"? Of course a four megabyte **AdRAM**™ **540** and **Flicker Free Video**™ with a multi–sync monitor. Why settle for less?

ICD

ICD, Incorporated
1220 Rock Street
Rockford, Illinois 61101
USA (815) 968-2228 Phone (800) 373-7700 Orders (815) 968-6888 FAX

Circle 2 on Reader Service card.

SAY REVOLUTION



The fastest growing video technology company in the world is looking for programmers to join our team. We've assembled the hottest development group in the industry here at NewTek. But we have two slots open for a new project that will blow your socks off. Have you always dreamed of working on revolutionary technology in a small, focused group? We need software innovators that want to create the products of tomorrow. Here are the skills you'll need:

- Strong 68xxx assembly-language programming skills
- At least 3 years of assembly-language programming experience
- Ability to write low-level code for time-critical applications
- Background in high-speed graphics and video applications
- Experience in programming prototype hardware
- Ability to quickly learn new custom chip architectures
- Background in low-level I/O and interrupt operations
- Intimate understanding of Amiga O/S and hardware
- Experience with video and graphics hardware
- Ability to read a schematic
- Strong organizational and project design skills
- Being a self-motivated, self-teaching, innovator
- An uncompromising drive for excellence

If you've got what it takes, you'll be forging ahead where no programmer's ever gone before. NewTek offers outstanding (and unusual), compensation and benefit packages for the chosen few who are a cut above. You'll work in an environment created by hackers, designed to be a hackers heaven. Wouldn't it be fun to invent things that are featured in USA Today, Rolling Stone, and TIME? At NewTek your brain can change the world.

Send Resumes to:

Alcatraz
C/O NewTek
215 SE 8th St.
Topeka, KS 66603

NEWTEK
INCORPORATED

Circle 4 on Reader Service card.