VLB ARRAY MEMO No. 387

840918

VLBA MEMO

From: Joseph J. (Jay) Johovich

Re: Monitor and Control; The View From The Top

The traditionly NRAO has produced telescope systems that have just enough computational equipment to make that system work. The designers at NRAO have done wonders at squeezing every inch of performance out of limited resources. For computer resources, the techniques of "Bottom-Up" design and the use of assembly language provided the most economical systems in terms of CPU cycle economics and address space economics, the two traditional factors for determining computer power. This design methodology also produces code that utilizes the unique elements of a particular CPU, memory system, and IO system, to produce a system "tailored" to that particular architecture.

"Bottom-Up" design using assembly language produces systems that fit the computational resources like a glove, but that glove fits only one hand. Expansion, equipment updates, and software maintenance are all much harder to accomplish in "tailored" systems. We are witnessing these problems now with the VLA. "Bottom-Up" design in higher level languages (FORTRAN) usually suffers from this same kind of inflexibility, but, to a lesser degree.

We've put up with these limitations of flexibility up to now partly because the cost of computers was much higher, and partly because we did not foresee the expansions that are now being implemented. We, also, underestimated the work required to achieve expansions (see VLA Computer Memo 172, "Whatever happened to the Pipeline?").

Low cost memory supporting large virtual address spaces, combined with low cost distributed intelligence have changed the economics of systems' design considerably. In today's designs, initial hardware costs are much less important because the costs for expansion, equipment updates, and software maintenance are an order of magnitude greater than those initial costs. This will be true for the VLBA as well. "Bottom-Up" conservatism just dosn't pay off any more. Initial results always look good, but the headaches downline aren't worth it.

As a "Bottom-Up" designer works up from the machine level code, solutions to different problems tend to present themselves. The designer tends to jump on these solutions in the context of the narrow constraints of the problem at hand. Because the designer has less of a feel for how these constraints might change with changes in the overlaying structures (due to evolution or expansion of the design), the solutions tend to be rigid, and soon the system starts

to "look" like the designer. In other words, the solutions that the designer tends to pick shape the system in such a way that the system mimics the designers style instead of modeling natural human systems. Everyone ends up having to learn that designer's style.

THE ALTERNATIVE

The alternative is to face the economic facts of today. We must specify a system that is modular and has provisions for expansion. We should weight CPU cycle economy at a much lower priority and memory use economy at an even lower priority. Finally, we should use a "Top-Down" design technique that stresses flexibility at all levels.

Actually, "Top-Down" is not the right euphemism, "Global-In" would be more appropriate. You start the design effort with all the goals, requirements, and limitations that apply. These include the scientific objective, available resources, and the design techniques themselves. Although this memo is primarily meant to discuss design techniques, there are many interesting factors in the other two categories that shape the kinds of "design techniques" that will be required. Thus, this "design environment" is indeed a "global", inter-relating environment.

This "Global, Inter-relating Design Environment" is used by the designer to guide the decisions he makes at each step in the design process. The designer works from this global environment towards the hardware core. He produces a design matrix that starts with the "Global, Inter-relating Design Environment" and ends at specific hardware functions.

Some of the more interesting factors involved in the VLBA Data Acquisition System's "Global, Inter-relating Design Environment" are:

1. Possible connection to non-VLBA antennas.
2. Possible rapid changes to any major module.
3. Availability of new "systems design" software and hardware.
4. Possible expansion to include the VLA, an OAE (Orbital Array Element), or the IBA (Intermediate Baseline Array).
5. Unknown future scientific requirements.
6. Unknown future hardware advances.
7. Newly emerging standards (IEEE 802.x, GKS, BitBus).
8. Possible use of software and hardware modularization techniques that will produce "generic" modules that can be used for new NRAO telescope projects, (like the OAE, IBA, or the MMA).
9. Possible extension of the above generic concept to include all telescope systems.

A possibility that protends a "standard" human interface to astronomical instruments.

10. ad infinitum...

Of course, the "Bottom-Up" designer also works within this environment. But, by the time he reaches the level where these factors are blatently important, he has invariably boxed himself in. The "Top-Down" designer has another problem. When he fails, it is usually at the point where he realizes that the hardware can't function up to his idealized model. In both cases it's "back to the drawing board" for the designers.

The "Top-Down" designer can protect himself by making sure that his hardware system will always be able to handle his design structures. This means flexible hardware and software. Unfortunately, extreme flexibility is always extremely expensive.

The only way to keep things cheap is to use common standards; standards that, through the magic of "volume manufacturing", produce cheap products and a pool of available talent to implement and maintain those products.

The connection between standards and flexibility comes when the "Top-Down" designer starts to modularize his system. A module is a collection of software and/or hardware functions that are logically connected in some way (i.e., they collectively serve some larger function). Modularization can increase flexibility if the interfaces between modules are well defined and the implementations of those modules are independent of those interfaces. This independence means that a module can be updated (a new algorithm or a new piece of hardware is substituted) without changing the interface; and, thus, not affecting the rest of the modules. As long as an improved module serves the exact function as the old module it will replace, the interfaces to that module are not changed. This stability means that the interfaces between modules are the most logical places to use existing standards. If high level interfaces are used, powerful standards can be employed which can support unexpected future needs. Standards can only be used for a subset of module interfaces, but this subset includes some of the most cost sensitive areas of the design: graphics, communications, and peripheral busses. The following standards (de facto and real) are candidates for these interfaces: GKS, IEEE 802.3(Ethernet), Multibus, BitBus.

Flexibility becomes important within a module. No apriori assumptions to module intelligence or future module expansion should be made. Thus, a module may start out as a "dumb module" in the design matrix, but it is not limited to that fate. If advances in radio astronomy indicate that

some new function requiring intelligence should be placed at that module, then it should be possible to do so easily. This feature recognizes the trend towards distributed intelligence and cheap micro-controller hardware.

The resources are available today to build this type of distributed, extendable system at a cost comparable to single processor systems of comparable CPU power. The real savings comes, downline, when functions are added or hardware is replaced.

Using the above approach, the concept of "loading" becomes much less important. "Loading" is the "Bottom-Up" designer's word for dividing up tasks between processors in a distributed intelligence system. Because of their bias, "Bottom-Up" designers tend to load processors such that each processor is used to capacity. That means dividing processor chores into chunks that fit the processors and have very little resemblance to "natural" divisions of the tasks. (Natural divisions occur in natural systems. Natural systems are systems, put together by humans, that are modularized in a way that is easily understood by humans, and, that modularization reflects the divisions put into other, similar and more familiar human systems.)

The "Top-Down" designer intentionally modularizes his system to reflect the "natural" divisions in the system. Unfortunately "natural systems" don't divide themselves up to be conveniently controlled by increments of processor power. A module where it would be "natural" to put a processor may not effectively use up a whole processor. And if a processor is used up by a module, that is bad because there is no room for expansion. In this last situation the "Top-Down" designer looks for additional natural divisions within the module (nested modules), so that the chores within that module can be distributed between additional processors.

Since every module will use different levels of intelligence, it might seem smart to use different processors with different levels of power for each module. Unfortunately, the real world of software maintenance steps in to shatter this idea. The alternative is to use flexible "processor systems". Memory, CPU's, and IO elements can be changed and expanded easily using these systems, but software needs only marginal changes. Of course, the system must be able to handle the most complicated modules; thus, some of the smaller intelligent modules will underwork even the minimum processor configuration. If a processor is sitting in a wait state for 20 minutes until an interrupt occurs, it is not committing a "mortal sin", especially if that lazy processor makes the system more reliable or easier to maintain and expand.

I think NRAO is having a problem with the semantics of distributed intelligence: "How many computers will we need?". Well, that depends on how you define computer. Larry D'Addario believes that our bus system is not a computer, although each of the IO cards used in the bus system has a CPU and ROM code, responds to IO, and could, if programmed, do much more than we are presently asking of it. In order for this not to be a computer, we must use it only as a bus IO card and we must cast the ROM code in iron.

But the specification of our bus system does include the ability to add functions. New system functions that require micro-controller level inteligence, that also require quick servicing turn-around times the standard bus system can't handle, that would require a small addition to the bus IO card's code or an aditional dedicated micro-controller, will appear. Under this scenario the ROM code would be changed and a new "bus IO card type" would be implemented. The result will be a growing list of "dumb modules" turned "smart modules" who must either talk "bus" (i.e., we maintain many bus drivers) or interface to the bus IO cards themselves (same problem, just twice as much work and twice the number of computers!).

Expansion using this rigid design is limited to the capability of the initial computer. Expansion requiring additional computer resources is the pits and the addition of distributed intelligence is even worse using this rigid formula. Instead a better way to view this system is to define two levels of intelligence. One level would be of the "microcomputer" scale and would be used for the larger scaled modules, would be few in number, and would be centrally located within the module's physical space (i.e., the telescope drive computer would be located in the telescope base, the tape computer would be located in the tape racks, etc.). A second level of intelligence would be of the "micro-controller" scale and would be a fully developed intelligence system (with high level language support, debugger, standard IO, etc.). This second level intelligence system can be located in any module and it will have many different activities to perform although a major function will be to handle communication of control and monitor data (among other things) between the level two systems and the level one systems.

This two tiered hierarchy of computational resources solves the problem of the lazy processor by allowing a whole different selection of processor power. A lazy module can be implemented using inexpensive "level two" intelligence, while busy modules can be implemented using "level one" intelligence.

Does a system exist that offers this kind of flexibility? Where two levels of processing power exist and standard interfaces between both levels and within the

levels exists both in hardware and software? Can it be true?

It is true. Intel's iAPXx86/310 series of computers combined with Intel's "BitBus" DCM system provides just such a system.

COUNTING COMPUTERS

If we use our present bus design we will end up with a bus IO card at up to 40 locations (more?). Every monitor and control module will have its own bus IO card. This sounds, at first, elegant, but is hard to implement. The multiplicity of nodes makes bus communication harder to control and time.

Of course, modules that require local intelligence or any other intelligent control (via communication with other modules) should always require a separate bus IO card. Modules that report monitor status only, or "report only" modules, should be divided into "clusters" (examples: all report only modules in one rack, or all report only modules in the receiver box, or all report only modules in the weather station). Each cluster should be serviced by one bus IO card. Connection of status bits and analog voltages would be delivered to the cluster bus IO card via a "physical connection standard". The clusters would be designed to allow these physical connections to be short, physically reliable connections.

A further reduction in the number of bus IO cards can be made by readjusting the scope of certain modules. No doubt Larry envisions a bus IO card in each separate video-converter module, ala MKIII. This degree of overkill actually reduces flexibility and reliablility. Since all the video converters must be grouped into a large video conversion module (for RF distribution ease, etc.), that larger super-module becomes a logical place for the bus IO card. If a physical connection standard (mentioned in the last paragraph) already exists, then connection of the individual VC's to the "level two intelligence system" is straight forward.

Using the above design criteria, the new total number of computers (using my definition of a computer), is less than half the number Larry envisions.

Bus traffic is reduced in my model because control is localized and commands are sent at a higher level. Also bus overhead is reduced because there are fewer nodes that must be polled.

LAST THOUGHTS

I should point out that the hardware I mentioned earlier can be used to implement a system very much like the one Larry envisions. Except for a few minor details (like differences in the minimum message size), BitBus does exactly what our bus does. If we use a single iAPX286/310 computer for each remote station and BitBus cards for the IO, the system will fit our present design formula at comparable cost. But, we can include the extra features provided by the full BitBus support system (languages, debuggers, standard IO, etc.). If we decide to design an "expandable" system, the above collection of hardware and accompanying software tools would be invaluable.

RELEVANT READING

Dahl, O.J., E.W. Dijkstra, and C.A.R Hoare, "Structured Programming", Prentice-Hall, Englewood Cliffs, N.J., 1972.
(The OLD TESTAMENT)

Myers, Glenford J., "Software Reliability, Principles and Practices", Wiley-Interscience, John Wiley and Sons, New York, N.Y., 1976.
(The NEW TESTAMENT)

Dijkstra, E.W., "Programming Considered as a Human Activity", "Structured Programming", and "The Humble Programmer", can be found together in: "Classics in Software Engineering", edited by E.N. Yourdon, Yourdon Press, New York, N.Y., 1979.

Parnas, D.L., "On the Criteria to Be Used in Decomposing Systems into Modules", Communications of the ACM, Vol. 5, No. 12, December 1972, pp. 1053-58.

Parnas, D.L., "Designing Software for Ease of Extension and Contraction", IEEE Transactions on Software Engineering, Vol. SE-5, No. 2, March 1979, pp. 128-137.

Britton, K.H., R. A. Parker, and D. L. Parnas, "A Procedure for Designing Abstract Interfaces for Device Interface Modules", IEEE Proceedings of the Fifth International Conference on Software Engineering, March, 1981, pp. 195-203, IEEE Computer Society Press.

Yourdon, E., "Top-Down Design and Testing", IEEE Tutorial: Software Design Strategies, 2nd ed., G.D. Bergland and R.D. Gordon editors, 1981, pp. 57-78, IEEE Computer Society Press.

Buhr, R.J.A., "System Design with Ada", Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1984.
(The Future!)