General Disclaimer

One or more of the Following Statements may affect this Document

- This document has been reproduced from the best copy furnished by the organizational source. It is being released in the interest of making available as much information as possible.

- This document may contain data, which exceeds the sheet parameters. It was furnished in this condition by the organizational source and is the best copy available.

- This document may contain tone-on-tone or color graphs, charts and/or pictures, which have been reproduced in black and white.

- This document is paginated as submitted by the original source.

- Portions of this document are not fully legible due to the historical nature of some of the material. However, it is the best reproduction available from the original submission.

Produced by the NASA Center for Aerospace Information (CASI)

UCID-30198
COMPUTER DOCUMENTATION

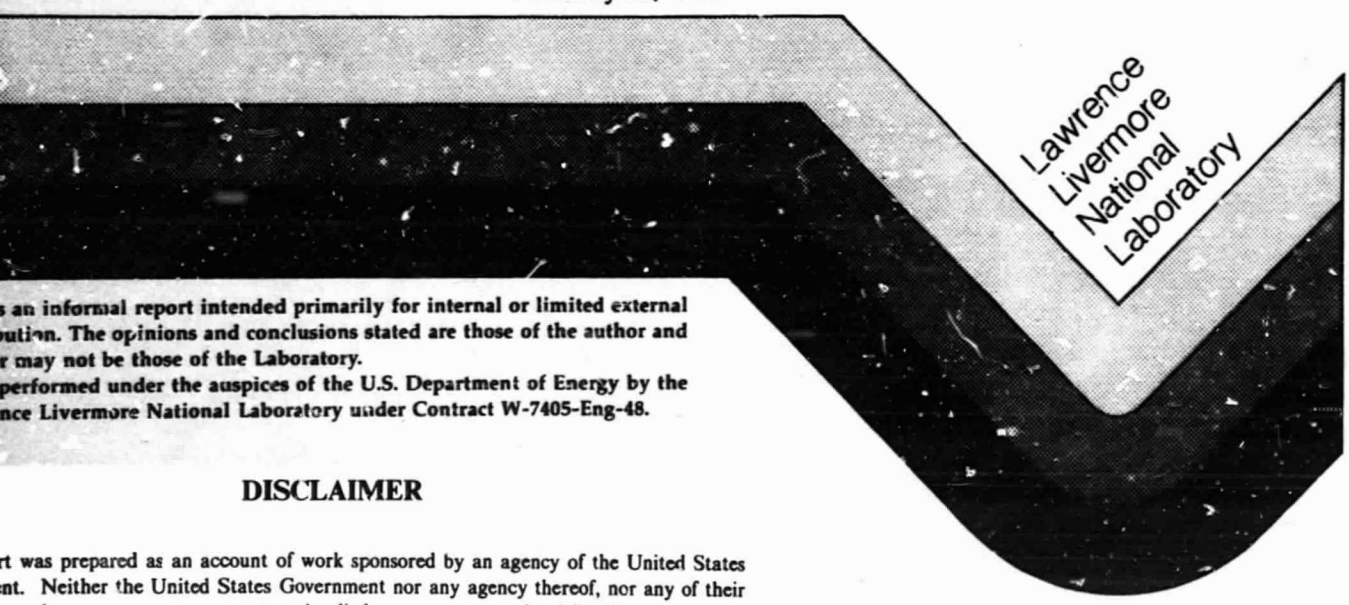
REQUIREMENTS FOR MIGRATION OF NSSD CODE SYSTEMS FROM LTSS TO NLTSS

UCID--30198

DE84 009533

Mike Pratt

February 22, 1984



**s an informal report intended primarily for internal or limited external
ution. The opinions and conclusions stated are those of the author and
r may not be those of the Laboratory.**

**performed under the auspices of the U.S. Department of Energy by the
nce Livermore National Laboratory under Contract W-7405-Eng-48.**

DISCLAIMER

rt was prepared as an account of work sponsored by an agency of the United States
nt. Neither the United States Government nor any agency thereof, nor any of their
, makes any warranty, express or implied, or assumes any legal liability or responsi-
the accuracy, completeness, or usefulness of any information, apparatus, product, or
sclosed, or represents that its use would not infringe privately owned rights. Refer-
n to any specific commercial product, process, or service by trade name, trademark,
rer, or otherwise does not necessarily constitute or imply its endorsement, recom-
, or favoring by the United States Government or any agency thereof. The views
ons of authors expressed herein do not necessarily state or reflect those of the
ates Government or any agency thereof.

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

CONTENTS

	Page
Abstract	1
Introduction	2
Background:	
The Implications of Change	
in The Nuclear Design Computing Environment	4
Overview of a Nuclear Design Code System	5
The Livermore Computer Center	
Viewed as a Computational Resource	7
Nuclear Design User Community	10
Why ND Computing Is Especially Sensitive	
to Environmental Change	14
Requirements	
for an Orderly Migration	17
General Requirements	17
An integrated computing environment	17
Maximizing turn-around	17
Command language	18
Controllers	18
Archival storage and retrieval	19
Output processing and delivery	20
Performance requirements and measurement	20
Accounting:	
Managing the delivery of the resource	21
Distributed processing	21
Multitasking	22
Software engineering tools	22
Documentation and training	23
Security	23
Specific Requirements	24
The dual-operating-system environment: LTSS and NLTSS	24
An idealized production code system: SIMPLE	26
A real code system	26
Specific functionality	27
References	30

ABSTRACT

The purpose of this document is to address the requirements necessary for a successful conversion of the Nuclear Design (ND) application code systems to the NLTSS environment. The ND application code system community can be characterized as large-scale scientific computation carried out on supercomputers. NLTSS is a distributed operating system being developed at LLNL to replace the LTSS system currently in use. We begin by examining the implications of change, including a description of the computational environment and users in ND. The discussion then turns to requirements, first in a general way, followed by specific requirements, including a proposal for managing the transition.

REQUIREMENTS FOR MIGRATION OF NSSD CODE SYSTEMS FROM LTSS TO NLTSS

INTRODUCTION

The purpose of this document is to express an overview of the requirements necessary for a successful conversion of the Nuclear Design community application code systems to the NLTSS environment. It does not purport to represent the requirements of the non-Nuclear Design Octopus user community. More detailed efforts to define requirements in specific areas of concern are underway (e.g., the "compatible" BASELIB/NLIB committee) and their work will be referred to but not reproduced here. This document is being restricted to the overview level in a deliberate attempt to make it long enough to be useful and short enough to have a reasonable probability of being read. Anyone with interest in more detail should go to the appropriate references. A lack of appropriate references may be an indicator of an area that needs work.

It is more traditional, perhaps, to deal with user requirements at the beginning of a project, rather than to attempt to formalize them after a project has been under way for a number of years. For this reason, and because of the difficulty involved in pinning down exact requirements for replacing such a large and rich computational resource as LTSS, we begin by providing a description of the Nuclear Design code environment for which NSSD is responsible, in the hope that this background information will assist the designers and implementors of NLTSS in making their daily decisions. The major point we attempt to communicate is that the ultimate users, the ND physicist community, have needs that are in many cases very much different from those of people who design and write programs.

In the next section we address the general requirements that must be satisfied to ensure an orderly migration of NSSD code systems to the NLTSS environment. Following that, we close with a section which briefly addresses specific functional requirements and sets forth recommendations on how to proceed from here.

We will attempt to address requirements in terms of functionality whenever possible and will also try to indicate the priorities involved, both in terms of timeliness and importance. In addition to describing what is required for the immediate conversion from LTSS to NLTSS, we will try to address future requirements in such a way as to make clear the distinction between current and future needs.

What fo ows is not complete and never will be, due to the complexity of the computational environment and the fact that it is a moving target. In addition, the Nuclear Design community is diverse enough to make it impossible to represent accurately. And, last but not least, a truly specific requirements document would take more time and the efforts of more people than we have available to devote to the task. We do intend to update this document periodically as new information is brought to our at ntion. None of the requirements is intended to be presented as carved in stone; there is always room for negotiation (where there's a will, there's a way.) Thus we believe that one of the major, and perhaps most important, purposes of this document will be to provide a focus for and a record of a mutually useful dialogue.

Please address feedback to Mike Pratt, L-16, ext 2-4699.

BACKGROUND:
THE IMPLICATIONS OF CHANGE
IN THE NUCLEAR DESIGN COMPUTING ENVIRONMENT

We are presenting a description of the Nuclear Design computing environment primarily to give the implementors of NLTSS an overview of ND's current usage of LTSS in order that they may draw their own inferences about requirements that have not been adequately covered.

The only previous efforts that we are aware of are:

An article in the October 1980 issue of *Computer*, written by Garry Rodrigue, Dick Giroux, and Mike Pratt (Ref. 1). Proceed directly to the last third of the article for the information that is most relevant here. Topics covered in this article include the modular composition of the large-scale scientific code systems in Nuclear Design and the potential distribution of functional elements. (Please note that this article was written circa 1978 or 1979 and was somewhat out of date by the time it was published.)

An article in the November 1981 *Energy and Technology Review*, written by Ed Woolery (Ref. 2). Ed is a weapons designer whose article is intended to cover the use of TMDS in his work. He gives a brief introduction to the work that a designer performs and makes some points concerning his use of the computational facility in general. Of particular interest are his conclusions 1) about such things as selecting the best resource for getting a specific task accomplished and 2) that the preferred mode of output is graphics, both production and interactive. He stresses that, from the designer's point of view, the purpose of Octopus is to provide computer resources for performing advanced computational physics and engineering calculations associated with nuclear weapons research.

All of which prompts us to put forth the idea that the user is relating to an Octopus "shell," in the Unix sense, which is composed of both LTSS utility routines and NSSD application code systems. The purpose of the present document is to represent the LTSS version of this shell so that it may be functionally replaced (and eventually improved upon) by the NLTSS shell.

Overview of a Nuclear Design Code System

A generic weapon-design code system is illustrated in Fig. 1. There are at least two methods used to control the execution of these systems. In one, all modules communicate via files and messages and are coordinated by a controller or master module. In the other, the modules are segments of code in memory, and communication is accomplished via the memory-contained data base. Most big code systems contain elements of both methods, with perhaps one or the other predominating.

The typical use of such a system proceeds in a cyclical pattern, as follows:

Setup (generation). The beginning of the cycle is problem set-up, which tends to be interactive and includes such tasks as the text editing of "generation decks," the retrieval and storage of archival files, and the viewing of output on the shorter-turnaround devices: terminal, TMDS, and RJET.

Preproduction (short, production-like runs): The generation phase is often followed by short production-like runs to perform parameter studies or see if the problem is going to behave as expected. Again, output is viewed on the shorter-turnaround devices.

Production run: Next comes the production run, which can consume hours of Cray time. This is usually done overnight or on weekends, but sometimes during the day because of schedule pressures. The most significant output is usually in the form of FR80 graphics on microfiche, and data bases which are stored in the central file storage system automatically by the production code via a microprocessor-controlled utility provided by NSSD. Also, production runs have been known to crash, requiring user intervention of various kinds before the run can resume.

Postprocessing: The output from the production run is then examined and postprocessed, another interactive task with output viewed on TMDS and, possibly, presentation graphics for programmatic reviews directed to the RJET and FR80. Often enough time has elapsed so that files must be fetched from the central storage facility to perform this postprocessing. (As the postprocessing data bases and tools have been improved over the years, the batch FR80 output from a production run has decreased in importance.)

Setup of new problem (cycle begins anew): Following the postprocessing phase, another problem on this or another code system will be set up, and the cycle begins anew. The next run is often a variation of the previous run on the same code system. When this process gets repetitious enough, a controller may be written (usually in BCON or COSMOS) to automate the entire set of phases — generation, production run, crash fixes, postprocessing,

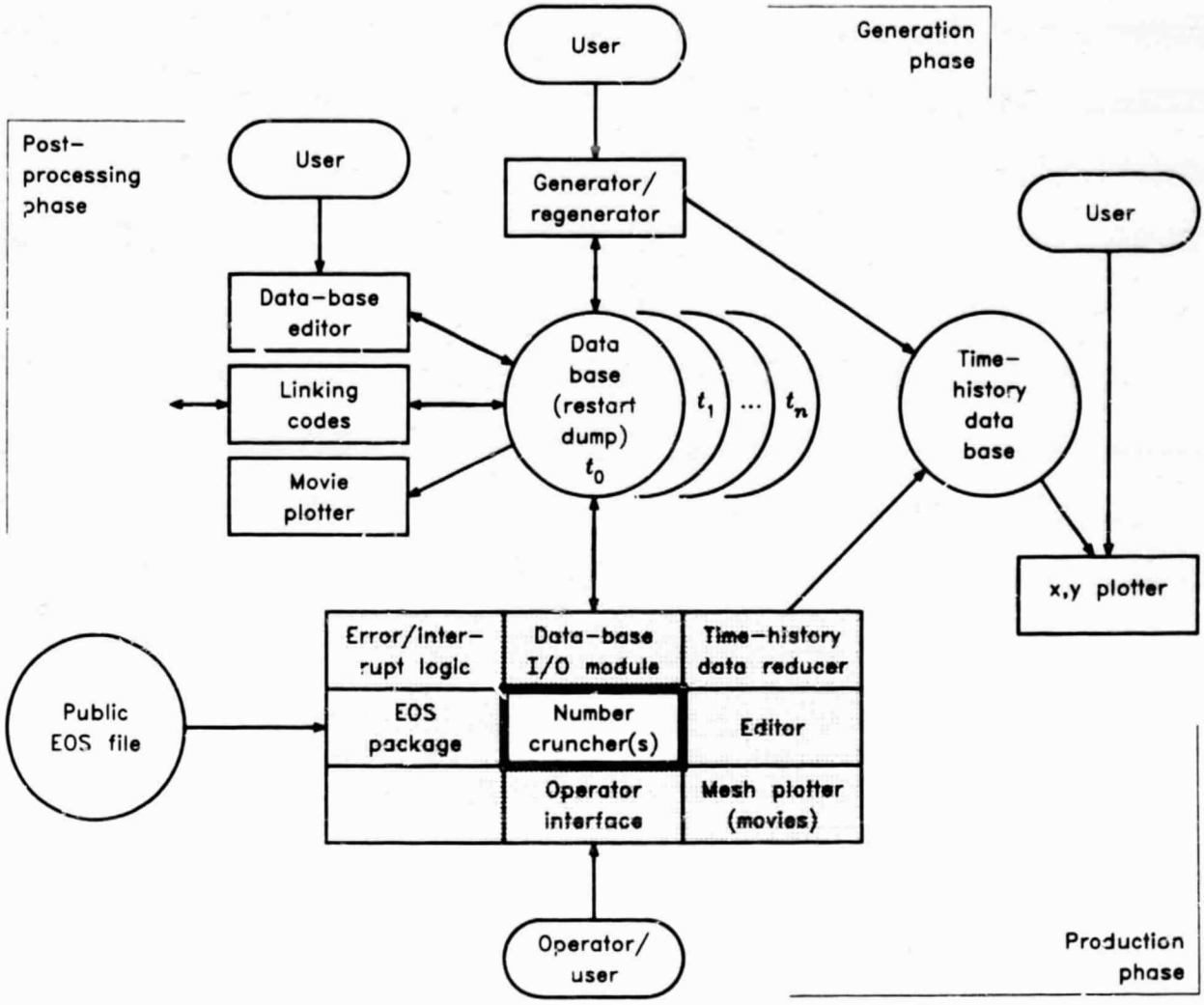


Fig. 1. The elements of a generic weapon-design code system and the phases of the cycle of use. The design code itself is the shaded block at the lower center, with the "number cruncher," or simulation physics module, as its kernel.

etc. This is ordinarily done only after the first few problems of a series have been run "manually" and a pattern has been established.

The Livermore Computer Center Viewed as a Computational Resource

The generic weapon code system illustrated in Fig. 1 does not rest directly on the supercomputer hardware. It is built upon what can be regarded as a computational resource consisting of the supercomputer itself, together with a concentric series of software shells. These shells, starting from the inside (hardware) and working out to the application, consist of an operating system, a programming environment (which itself consists of subroutine libraries and utility routines), and an application programming environment (which consists of application subroutine libraries and utility routines). This concept is pictured in Fig. 2, much as it exists today. Together these shells comprise a virtual computer, or computational resource. This resource is a tool that is used by the Nuclear Design weapons designer. Detail on this subject can be found in Ref. 2, which was written by a weapon designer.

One of the things that Fig. 2 illustrates is that building an application environment that makes use of more than one computer involves working with the operating system and programming environment that surround each computer. One can see from this model why it might be better to have a distributed operating system such as the one pictured in Fig. 3. In this model, the builder of an application environment would be working with a shell that assumed the burden of connecting the computers in an integrated fashion. Instead of having to deal with the separate programming environments surrounding each computer, one could deal with a single, integrated programming environment. In principle, at least, the prospect sounds attractive.

Since NLTSS is a distributed operating system, one can begin to see what is involved in making the transition from LTSS to NLTSS.

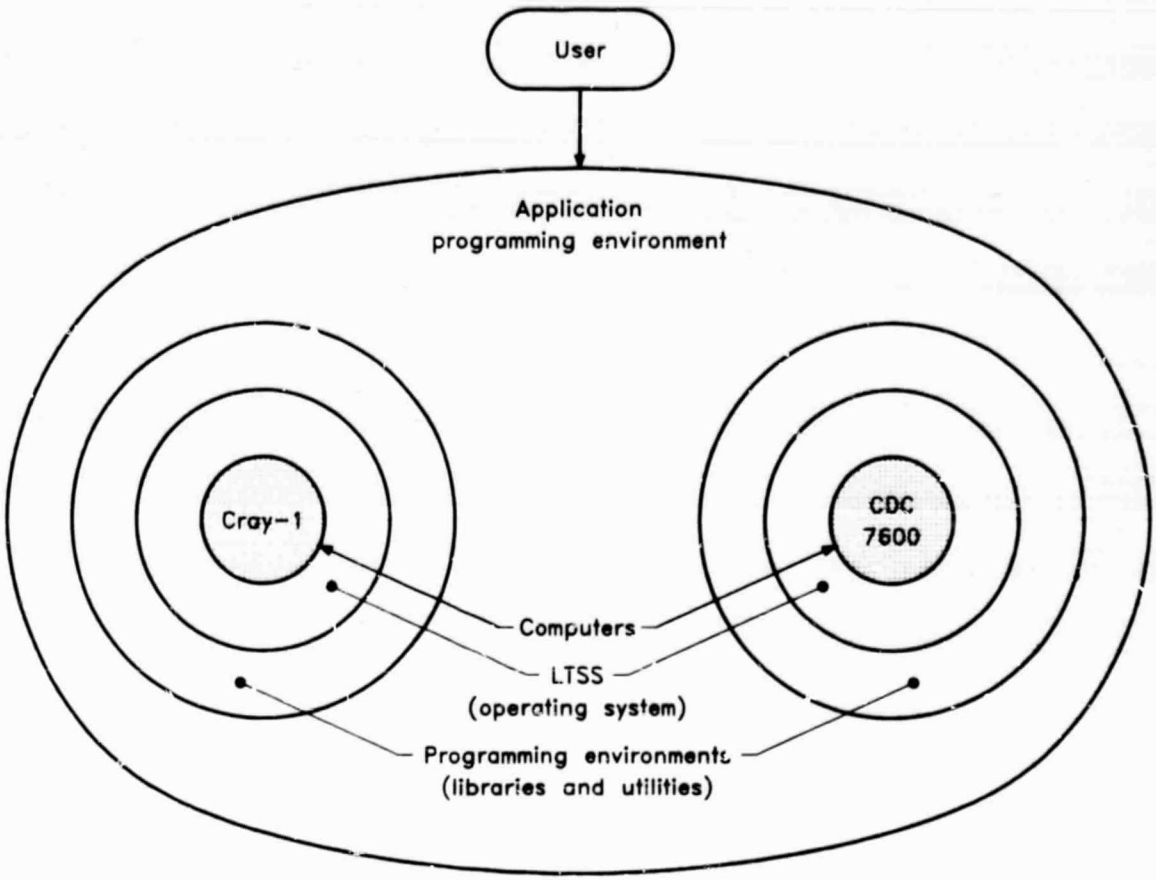


Fig. 2. The Octopus "shell," with a separate programming environment for each computer.

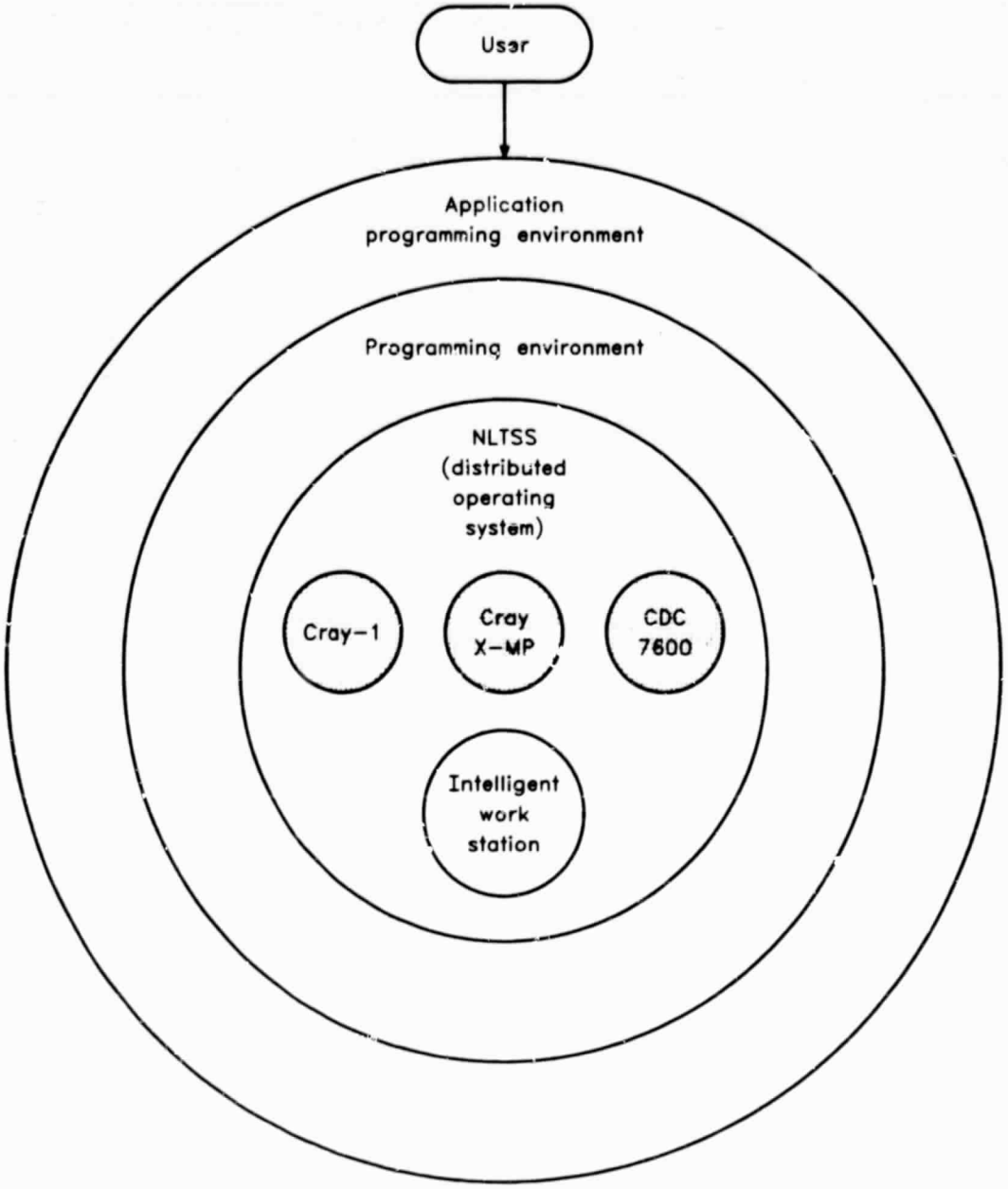


Fig. 3. The NLTSS "shell," with a single, integrated programming environment.

Nuclear Design User Community

The first element shown at the top of Figs. 1–3 is the User. Here we will briefly describe who these users are, what they do, and how they relate to the system.

From the point of view of the systems programmer designing and implementing NLTSS (the tool-builder), everyone in the ND community may look like a tool-user. Within the community, however, the users fall into three general types, occupying different regions of the spectrum between the purely tool-building and the purely tool-using. These three types are the applications programmer, the code developer, and the weapon designer. Their positions on the spectrum are illustrated in Fig. 4.

This is not meant to imply that the functions of the various categories are strictly separated. Figure 5 is an attempt to present their actual interrelationships. The communication interfaces between the four groups are shown, as are the overlaps in the roles of physicist and computer scientist — where code developers are doing applications programming, applications programmers are doing systems programming, and so on.

Designers

Of the three categories of Nuclear Design user, the designer is the ultimate customer. Designers do far more tool using than tool building.

When designers are using the computers, the entire code system shown in Fig. 1, together with a number of other similar code systems and many utility routines, comprises their set of tools, and the cycle of use described for the code system also describes the designers' work pattern. This working environment is extremely large, rich, and complicated, and not particularly well integrated.

In addition, designers have other things to do besides tending large computer calculations. To free themselves to do these other things, they have come to rely heavily on the use of software controllers as surrogate users or operators. The designers often write these controllers themselves, in COSMOS or BCON, or they may use prepackaged controllers such as OPTCON or LASCON. In the near future some of these controllers may reside locally on intelligent terminals.

The designers' primary focus is on getting the job done, not on keeping up with the latest in computer science. Hence designers tend to be very conservative regarding changes in the environment: they are skeptical about changes that are claimed to offer them long-term benefits at the cost of short-term trauma. They derive little comfort from being told that great

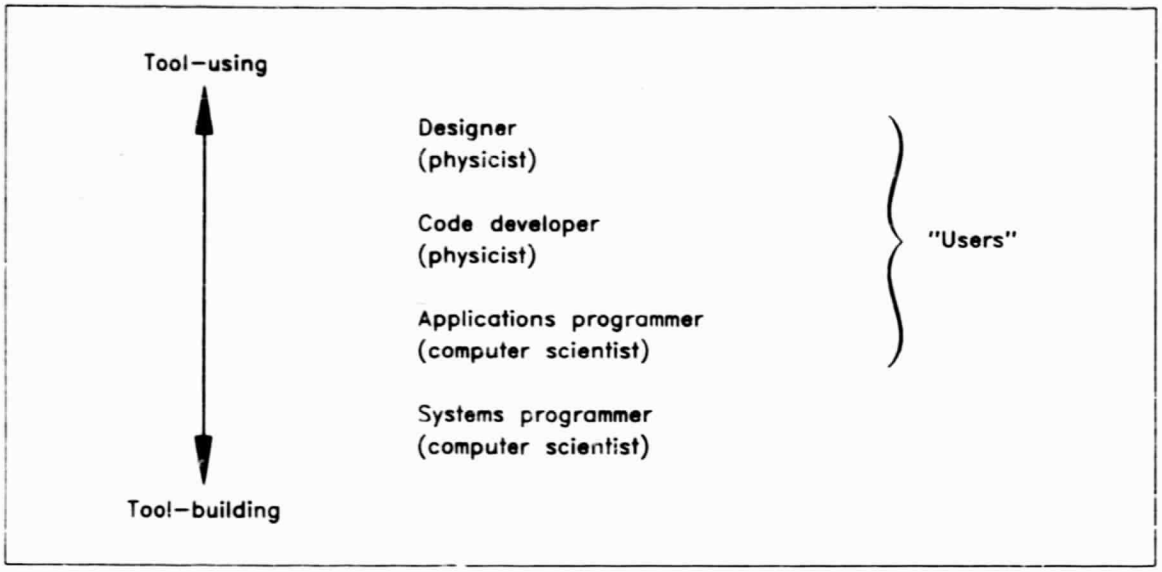


Fig. 4. Spectrum of tool–building to tool–using. The systems programmer, who provides the tools to the user community, is nearest the tool–building end. The Nuclear Design user community is made up of three types: nearest the tool–using end, the weapons designer, who is a physicist; next, the code developer, who is also a physicist; and next, the applications programmer, who, like the systems programmer, is a computer scientist.

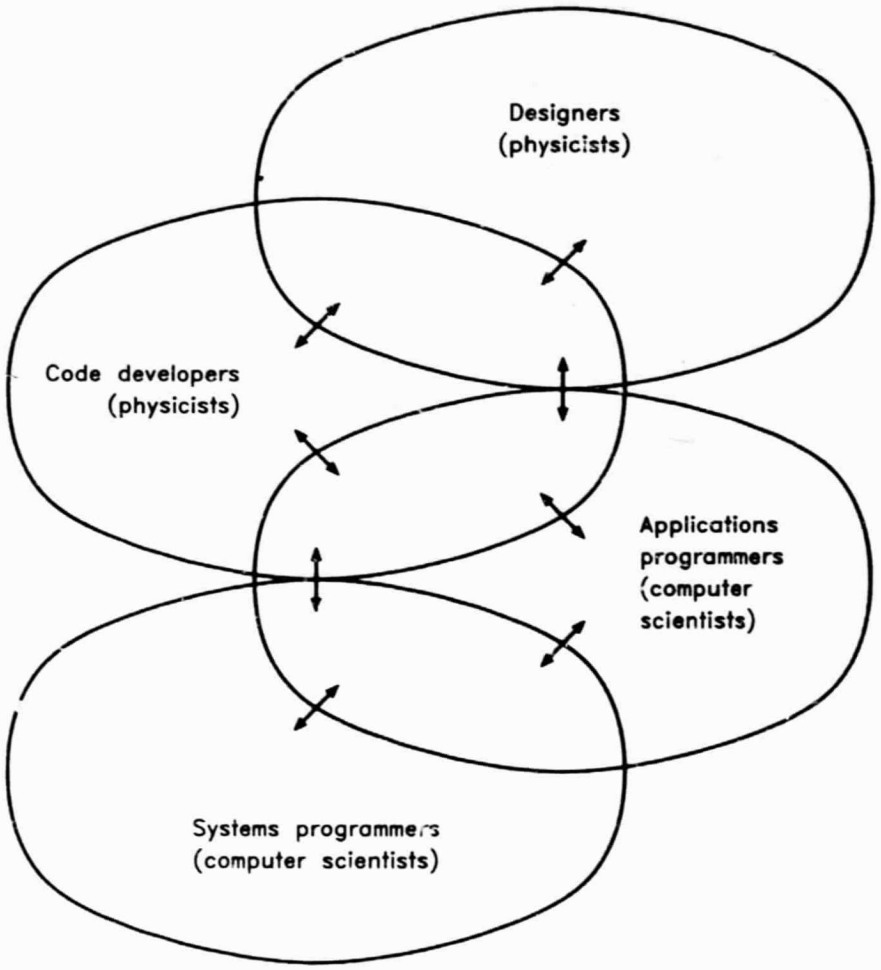


Fig. 5. Communication interfaces and overlapping roles. The functions of the providers and the users, and of the various types of user, are not strictly separated.

new changes are on the way that will make their lives easier eventually, but in the meantime they may have to make some sacrifices — they could lose some functionality, turnaround will be degraded, the computer environment may not be as robust, and in the end they will have to learn a new way of communicating with it. They might well wonder whose lives are being made easier: the tool users' or the tool builders'? To be motivated to learn how to use a whole new set of tools, designers must perceive demonstrated benefits. Given the demands of what they do, it would be unreasonable to expect them to feel any other way.

Code developers

Code developers bear the overall responsibility to Nuclear Design for the code systems assigned to them. In the execution of this responsibility, they are directly responsible for the kernel of the code system — the "number cruncher" or, more properly, the simulation physics module of the system shown in Fig. 1. Interfacing between the designers and the applications programmers, code developers are both tool users and tool builders. In their role as tool users, they behave much like designers: they generate problems, run them, and postprocess them as designers do (although for different reasons). In their role as tool builders, they can work a lot like applications programmers, including doing a great deal of their own programming.

Applications programmers

Applications programmers are responsible for building the shell which surrounds the physics kernel in the system shown in Fig. 1. It is this shell that renders the kernel useful to the designers. In addition, applications programmers are responsible for the interface between the kernel and the applications shell, and they often program some or all of the kernel itself. Thus, applications programmers, while they are tool users, are further toward the tool-building end of the spectrum. (They are actually using tools of a different kind, also — e.g., libraries of subroutines rather than utility routines and code systems.) In this position, they tend to be more cognizant of the computing environment as such, and of the need to improve it (and therefore sometimes to change it). Nonetheless, systems programmers should bear in mind that, as users, the applications programmers have a job to do and that this often causes them to take the point of view of the physicists. But applications programmers are also computer scientists, at the interface between systems programmers and the physicist community; they are, therefore, often pulled in two directions at once.

Conclusion

In summary, the major point we are trying to make is this: As tool users, designers' work differs markedly from that of code developers, application programmers, and systems programmers; and since their work differs, their requirements differ also. It will not be satisfactory, therefore, to build a new computational environment that only satisfies people who spend most of their time programming.

Why ND Computing Is Especially Sensitive to Environmental Change

For reasons having mainly to do with size, complexity, and long lifetime, the Nuclear Design code systems themselves are very difficult to change, and their use can be greatly perturbed by a change in the environment. Some of the features responsible for this sensivitivy are the following:

- Many modules: ND code systems are made up of many different modules (the rectangles in Fig. 1): subroutines, segments, controllers, and controllees. It can take days, weeks, or months to make and test a major change. Thus, making changes is not a process entered into lightly. Working in this environment places the applications programmer and code developer at the mercy of many people, some of whom they have never met.

- Communication between the modules: A major feature of the big application code systems is the method of communication between the various modules in the system. There are two major means of communication: messages and files. (There is also intermemory communication within a controllee and its segments, but we are not concerned with that here; or are we?) Requests made of the operating system tend to involve one or the other of these two means of communication. An operating system which moves file and message traffic slowly or unreliably will negatively impact the user's ability to get the job done.

- Many libraries: Nuclear Design code systems tend to be very large and load with many libraries — sometimes 10 or more. A representative hierarchy of a code within the operating system is shown in Fig. 6. These libraries really form part of the higher-level language in which applications programmers work. They are used for accessing data bases (ACFLIB), interfacing with the operating system (BASELIB), performing formatted I/O (FORTLIB, BLIB), dynamically allocating memory (MMLIB), drawing pictures (GRAFLIB, PLOTLIB), and other specialized tasks (MATHLIB, STACKLIB, EOSPLIB.)

- Long lifetimes: Most of the big code systems have been under development for years; some go back decades. A few of the application system modules are old ones that have not been recompiled for years, even

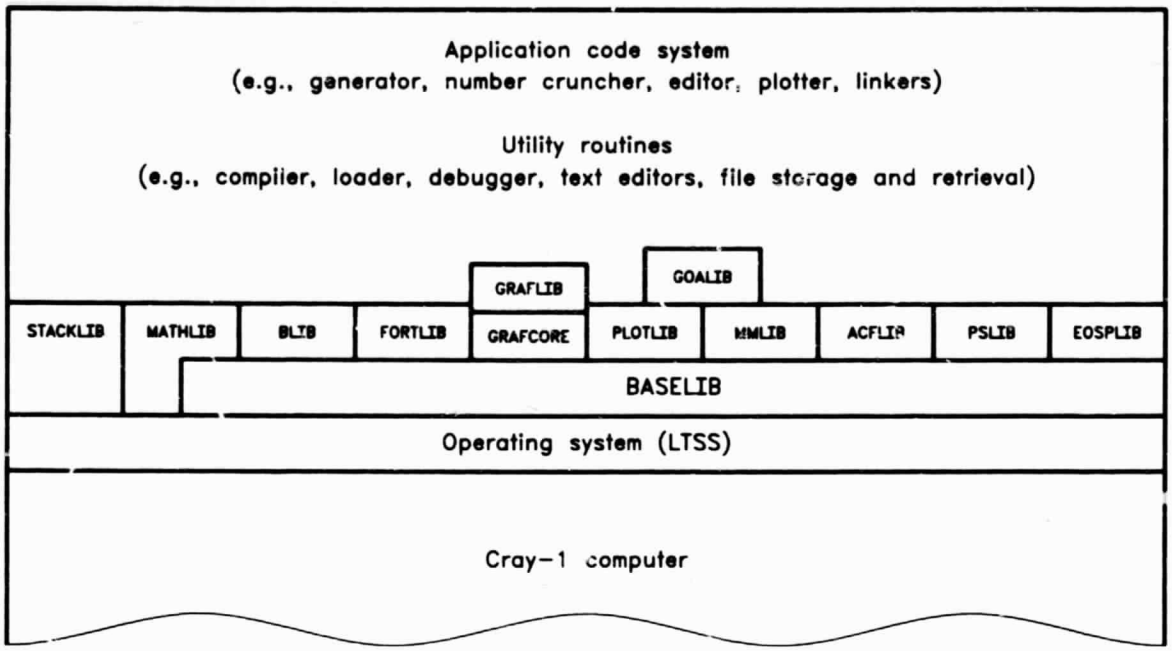


Fig. 6. Nuclear Design code systems often load 10 or more libraries.

though they are still being used. And when they are recompiled or reloaded, they are often being "maintained" by some one who did not write them and does not have the time to thoroughly understand them.

In the light of age alone, it is somewhat unreasonable to announce that some important subroutine library is being updated next week, and that everyone should reload to make sure they are not affected.

In fact, given the number of libraries that most of these codes load with, a fair number of applications programmers could find permanent employment just reloading the entire inventory to make sure nothing is affected by the latest updates. To avoid this kind of overhead, most codes are reloaded only when they are changed; and when things go wrong, as Murphy's Law says they will, the debugging task includes identifying which libraries and other tools have changed since the last successful version. (The debugging task is often further complicated by the fact that there are numerous private versions of almost everything.)

REQUIREMENTS FOR AN ORDERLY MIGRATION

General Requirements

This section lists the general features NLTSS should provide to enable the Nuclear Design code systems and user community described above to successfully convert from the old environment to the new. Specific recommendations regarding the transition procedure and requirements with respect to languages, utility routines, and subroutine libraries are dealt with in the next section.

An integrated computing environment

NLTSS should present ND designers with a complete, integrated environment that works, and works better than the one they are accustomed to.

Designers must learn how to communicate with four or more application code systems and a large fraction of the utility routines in the Summary Sheets, frequently write their own controllers, and sometimes do their own programming. This is made more difficult if each application code system and each utility has a different set of commands and follows a different file-naming convention. Hence integration of the computing environment would greatly facilitate the designer's job, while lack of it is a decided hindrance. (Reference 3, a memo written by two physicists new to Nuclear Design, expands on this theme with a number of detailed suggestions. All of the views expressed are not necessarily endorsed here; it is the level of concern expressed that demonstrates the problem.)

Maximizing turn-around

Because running the big design codes is so time-consuming and scheduling pressures are often great, turn-around is important to the designer. In order to keep jobs moving, designers often come in on the weekend or in the middle of the night to examine the output from one production run and set up the next. Listed below are some of the factors which affect turn-around.

Tools to help decide which resource to use: Up-to-the-minute information on the availability of resources (e.g., the TMDS status displays) is essential to enable the designer to make the best decision on which resource (e.g., Cray, RJET, etc.) to use for a given task.

Distributed, remote output: Output devices such as TMDS and RJET have become indispensable in furthering turn-around. They have also reduced the

volume of material that would have otherwise been sent to the central output facility for processing by the FR80s and high-speed printers.

Messages and files: Since these are the two major means of communication between the modules in the large application code systems, it is essential that an operating system be able to move message and file traffic quickly and reliably.

Command language

It is our understanding that the initial Command Language Interpreter (CLI) will look like LTSS (i.e., provide an LTSS-compatible shell.) We need an explanation of what this means, in terms of a model to which the users can relate, to allow them to begin making effective use of the new shell. Pete Du Bois discusses this subject in his article in the August 1983 issue of *Tentacle* (Ref. 4).

An important point that we wish to reiterate is that designers do not have the same requirements as code developers, application programmers, and especially systems programmers. For example, designers spend a lot of their time managing multiple production runs and thus have need of tools that will assist them — such things as windowing or improved suffix capabilities for keeping track of three or four jobs running on different main frames, and directory and output processing capabilities that will help them keep track of their voluminous output.

Bob Cooper is the NSSD representative on the CLI Design Committee that will be working on the new, improved CLI.

Controllers

The use of controllers by designers is mentioned above, in the section describing the ND user community. Among the examples mentioned were the use of controllers to coordinate the execution of the modules that make up a big code system, the creation of controllers to automate repetitive procedures, and a microprocessor-controlled utility that stores files for the big code systems (this is discussed further in this section under "Archival storage and retrieval.") Before designers will transfer to the NLTSS environment, they will need to have been provided at least the same functionality that they have now. The macro capability in IMP has been used by many designers (and other users as well) to build the equivalent of superfunction keys to interact with utilities and application code systems.

The subject of NLTSS controller/controllee relationships (e.g., message bypasses, the strictly linear or stacked relationship between controllers and controllees) has proved to be a troublesome issue for the NLTSS implementors. Our understanding of current plans is that an NLTSS server or process will be implemented to simulate the present relationships. This is vital, and it must be reasonably efficient. The irony of this situation is

that the capabilities of NLTSS are being constrained severely in order to provide this strict simulation.

There will also be the need for production control tools such as MICROCOP, SUPERCOP, and PAD. Exactly what that will involve in the NLTSS environment is not clear to us at this time. This subject is entwined with the concerns mentioned in the Accounting section below, as these tools have been developed to manage the delivery of the computational resource.

Archival storage and retrieval

Some of what is still out there in the real world was designed a long time ago — it is not unusual for a designer to have to recalculate a decade-old design. Starting from scratch in such circumstances could take months; the alternative is to be able to resurrect or recreate old codes and retrieve old data bases. Hence designers need long-term, archival storage and backward compatibility of the data files used by application code systems.

Also, as was mentioned earlier, designers have to deal with enormous amounts of output. They need an archival storage and retrieval system that will give them maximum assistance in managing all this data. A very important point that is often overlooked or misunderstood is that, while they will never again look at the majority of the data they have filed away, they cannot predict in advance exactly which fraction they will need in the future.

Mike Pratt generated a user requirements specification, dated 11/18/80, for a file management utility that covers the requirements of the Nuclear Design community in some detail (see Ref. 5). Perhaps the single most important issue raised in this document is that of reliability, where reliability is defined in this context as the instantaneous willingness to assume the responsibility of accepting a file for storage, no matter what. A production code system cannot come to a halt while waiting for ENTERPRISE to come back up; it must have some reliable (as defined above) means of getting its files stored. NSSD has created a utility named BPORT to do this. We have suggested that it would be better if this functionality were provided by the file-management utility itself instead of being grafted on by users.

Also, Garrett Boer headed up the Archival Storage Overload Committee which produced recommendations in late 1982 (Ref. 6). (Linnea Cook and Mike Pratt were members of that committee.) We are also interested in efforts to redesign ENTERPRISE and the directory system.

It is our understanding from talking to Pete Du Bois that the file system will be integrated with NLTSS so that it is, in effect, a continuum. We need an explanation of what this means in terms of a model that the users can relate to. For example, it may make no logical difference to the operating system where a file is when access to it is requested, but there

is a practical impact on the requester, whether it be a user or a code, in terms of when the access can be expected to be granted and how long it will take to make each subsequent access. It also seems clear that there will be a need for new and improved directory manipulation capabilities to perform such tasks as sharing resources, searching for files, and so on. This might be a good time to reopen the question of providing a way to satisfy some long-expressed user requirements (e.g., to be able to find a file named "zig1234" that a user knows (or hopes) is filed somewhere in the system).

Output processing and delivery

Under this heading we include CHORS, CHORS II, TMDS, RJET, NIPS, FR80, and maybe even intelligent terminals and local area nets. Issues include quality assurance (finding problems before the users do), accountability (Where is it if it's not in my box? Who gets charged for it? What went wrong? Who's responsible for fixing it?), and security. The processing of output files should also be addressed — e.g.: How many file extensions will be allowed? What will be done when the maximum number (if any) of extensions is reached during a long production run? How many files will there be in a family (if families are necessary or even possible)? How will we keep "broken" fiche problems from arising?

We also need to do a better job of characterizing the output of the big production codes in terms of volume. Gary Henderson of NSSD has been leading an effort looking into methods for gathering I/O statistics from NSSD production codes.

Where will NLTSS end and the output system begin, from the user's point of view? We need an explanation in terms of a model or models that designers, code developers, and applications programmers can relate to.

Performance requirements and measurement

Chris Hendrickson has indicated that he will be coordinating efforts in this area. Speed and size issues are very important to Nuclear Design codes (and difficult to quantify). A crucial issue will be the response time of the LTSS-compatible shell relative to the response time of the environment it will be replacing. It should not take significantly longer to perform any regular task.

Now all we have to do is define more precisely what the words "significantly" and "regular task" mean in the previous sentence. Does anyone out there have any data that would help (e.g., a study that shows that some measured degradation in response time was not considered significant by some population of users)?

Accounting:
Managing the delivery of the resource

Nuclear Design needs to be able to manage the delivery of computational resource in a timely manner. There are too many users spread over too many physical locations for the informal tools of yesterday to suffice. Tools have been developed in Operations and within Nuclear Design to collect and massage data in this area, and they will have to be modified or replaced. Some of the tools (e.g., MICROCOP, SUPERCOP, SNIPER, WATCHDOG, NATES, PAD system) are used to control the delivery of the resource, and some (e.g., TCSM, ATIME, COLLECTOR, SNOOPY) are used to report on the delivery of the resource, often the next working day. The portion of all this that belongs to NSSD runs to more than 23,000 lines of source on the CDC 7600's and Cray-1's and 3700 lines of source on the ND microprocessors. We believe that FRAMIS is currently being used in parts of this effort by COD. Requirements in this area are important enough to be addressed now and satisfied as NLTSS is implemented, in an integrated fashion, not added on later.

Distributed processing

An important source of information regarding the potential distribution of ND application code system processes to a near support processor (defined as a midicomputer or minicomputer working in close association with a maxicomputer) is contained in a study committee report produced by J. Fletcher, A. Leibee, J. Minton, and J. Randolph, dated January 1979. (See Ref. 7.)

Interprocess communication: We will define distributed processing as spreading the processes or modules of an application code system over more than one discrete computer. Since NLTSS is an instance of a distributed operating system built on the LINCS (Livermore Interactive Network Communication Structure) network architecture, all that is left are the details. Applications programmers will want to use the interprocess communication facilities to provide new functionality that will act as an incentive for designers to work in the NLTSS environment.

Local area networks (LANs): As LANs with increasing amounts of local computing capability are implemented in the Nuclear Design community, this will become an increasingly important subject. For example, it is hoped that in the not-too-distant future, significant mesh generation and manipulation capabilities can be offloaded from the main frames and onto reasonably powerful local scientific work stations. Again, LINCS appears to hold promise as a tool to help bring this about.

Multitasking

We will define multitasking as one program making use of two or more CPUs that have access to a common memory. To quote Pete Du Bois, "NLTSS is designed and implemented from the ground up to support multitasking in a natural and efficient way." This is a vital area since virtually all new supercomputer designs include multiple CPUs as a way to gain improvements in performance. The immediate future will be devoted to learning how to do it, with emphasis on techniques such as forking and joining, managing shared data space, multiple copies of subroutines, and code reentrancy.

Software engineering tools

A laborious task for code developers and applications programmers is the debugging of modules that must be reloaded because either they or their support libraries have been changed. The section detailing why ND computing is especially sensitive to environmental change (p. 14) provides much of the justification for making this statement. This task would be made much easier by the availability of software engineering tools that address such topics as change control, environment inquiry and retrieval, and updating of public files, together with effective quality control over such topics as release testing and validation.

Debugging tools: The impact of the functionality contained in DDT on the productivity of code developers and application programmers cannot be overemphasised. Any regression from such capabilities as symbolic debugging and breakpoint setting would cause orders-of-magnitude degradation in productivity. Seemingly minor issues can assume great importance also; for example, a dropfile may be all that remains (not counting restart dumps, incomplete output files, etc.) of a many-hour run that died unexpectedly (together with a garbled rendition by the user of what appeared on the terminal.) It is unreasonable to mandate that such a job must be rerun for debugging purposes, which means that the dropfile functionality must be retained in some form.

Environment saving/restoring capability: This refers to the discussions that NSSD people have been carrying on with USD people and also to the work that Gene Albright has done. As mentioned above, when a programmer has trouble with a code it would often be useful to be able to re-create the environment in which it was originally compiled and loaded, especially with respect to all of the subroutine libraries that were used.

Network issues: To be included here are such topics as naming conventions for systems resources. More to come as we gain experience.

Feedback: What applications programmers desire from an operating system — in addition to functionality, efficiency, and reliability — is responsiveness or feedback, e.g., intelligible, accurate error messages. Applications programmers spend a lot of time and effort trying to figure out

what happened and what to do about it, and they need all the information they can get as quickly as they can get it.

Quality assurance: Issues include those concerning output quality and reliability mentioned above, plus software release testing and quality control.

Documentation and training

Issues include the training of applications programmers, code developers, and designers as well as consultation and assistance for users. Material should be developed to aid in motivating users to adapt to the new environment. There will be a need for complete documentation, and, most important, "summary sheet" documentation, since that form of documentation will be the only one the majority of users will read.

Security

Chuck Cole is heading up the Computation Department Security Committee, which is working to explore, define, and make recommendations for resolving issues in this specific area. The committee has met with designers from the ND community.

We must remember that many of NSSD's users spend a considerable portion of their time dealing with and running at level 5. Important issues include:

- Oversight and recovery (e.g., master keys for systems managers to allow them access in the absence of key personnel.)
- The security of possible links into Octopus from outside the Q area.
- Creating files with different security levels from the one you are running at ("writing up" and "writing down").
- Should the classification of graphics output be a file- or a frame-level attribute?
- Changing your operating level after logging on, or running jobs at different levels simultaneously.
- Classified file and directory names.
- User-in-attendance requirements for remote output devices such as TMDS and RJET.
- The methods for handling, processing, and delivering classified and unclassified output.

- A general principle: Make it as easy as possible to follow the rules, so that people won't be tempted, much less forced, to break them to get their job done.

Specific Requirements

Now that we have spent some time describing what the computational environment is like now and what it should be like in the future, we need to get down to business. A more appropriate heading for this section might be "How Do We Get There From Here?," where "there" is the improved computational environment represented by NLTSS. The guiding philosophy will be to first replicate what is available now and then to improve upon it.

The dual-operating-system environment: LTSS and NLTSS

To aid in this discussion, we need to refer to the model presented in Fig. 7. This model represents our current understanding of the "dual environment" being implemented as part of the NLTSS project. In this dual-operating-system environment, a Cray-1s or Cray X-MP is partitioned into two logical or virtual machines, one devoted to LTSS and one devoted to NLTSS. Each of the two virtual machines will have resources such as memory, disks, etc. allocated to it and it alone. That means, for example, that a file created under NLTSS will not be accessible under LTSS and vice versa. (Ref. 4 says that there will be a utility provided to move files between environments.) Also note that each disk unit will be allocated to one of the operating systems.

Migrating a code from LTSS (Step 0) to NLTSS (Step 1) will ideally involve nothing more than reloading with NBASELIB and NLIB. This will place the code in the LTSS look-alike environment, where one cannot tell, in theory at least, that anything is different. If that is going to be the case, why bother — why not just stay with LTSS?

In fact, there will be differences, some small and some large. For one thing, interactivity will probably be worse, since NLTSS will have to be artificially constrained to make it look like LTSS. An example of this sort of restraint will be the existence of the chain server depicted in Fig. 7, which will exist only to reproduce the LTSS controller/controllee relationships that we have all come to know. These relationships are a rather severe limitation of the more general process-to-process relationships that NLTSS will make possible. Implementing these limitations will probably be costly in terms of resources.

Since all of the differences mentioned so far are negative, none of them is likely to encourage one to make Step 1. The sole reason, in fact, for making this step is that it will position one for the future. Once a code is operating in the LTSS look-alike environment, it is really sitting

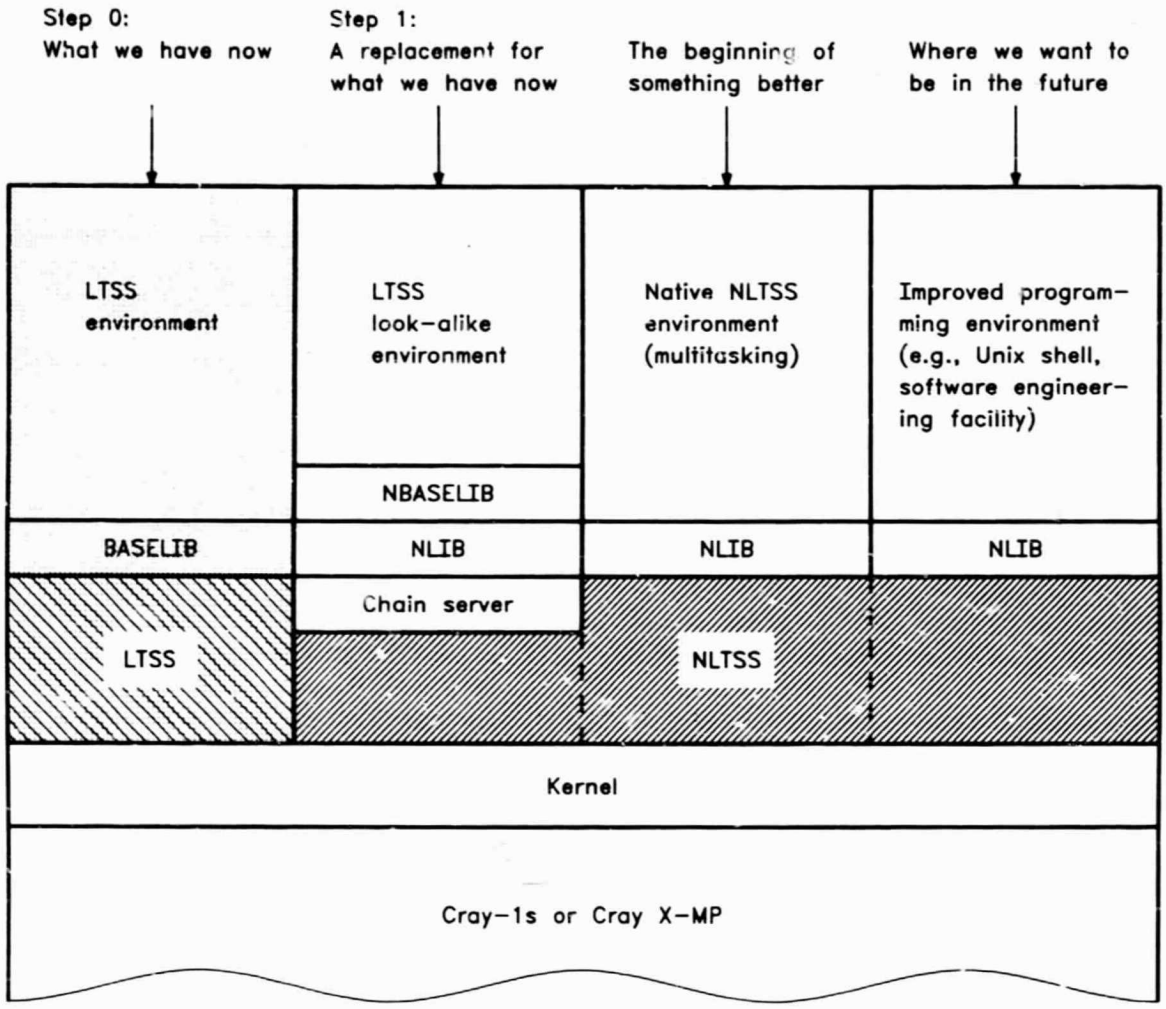


Fig. 7. Steps in a phased transition.

on top of NLTSS, which will make it possible to begin exploring the capabilities of the "native" NLTSS environment depicted to the right of Step 1. One of the most attractive short-term incentives will likely be the capability of multitasking (i.e., making use of more than one CPU by one code) on the X-MP, which will be available only under NLTSS. Another possible short-term incentive might be the ability to make use of machine time that has been allocated to the NLTSS partition on a given machine. The long-term incentive will be the migration towards the improved computational environment depicted on the far right of Fig. 7. Included in an improved environment of this sort could be such things as a Unix or Unix-like shell for code development. (See Ref. 8 for a discussion of this possibility.)

An idealized production code system: SIMPLE

We are proposing that a simplified, idealized production code system based on the SIMPLE code (Ref. 9) be provided by NSSD to be the first to make Step 1. This proposal has a number of advantages, the first of which is that it would involve only applications and systems programmers (no user demands!). Since only computer scientists would be involved, communications problems would be reduced and the physicist community would not be exposed to a new, and therefore unreliable and untuned, environment. This phase could start when a minimum of functionality was available, and there would be room for negotiation and compromise about what tools would be necessary and when. The SIMPLE code system could be beefed up to include most, if not all, of the range of production-code system tools and techniques.

The use of an idealized production code system would also afford the advantage of working with a stable application and without the pressure attendant upon working with a "live" production code system. If the idealized system is sufficiently modular, runs could be set up to test specific NLTSS functions before others are ready. The use of this system should also allow serious tuning of NLTSS for efficiency before the pressure of "live" usage is faced. Only after as many problems, bugs, and inefficiencies as possible have been addressed in this manner should a real production code system be moved, a step which would involve both code developers and designers.

A real code system

Moving a real production code system would involve code developers both because they would be required to find "physics" bugs and because they often do their own physics programming; once a code system is moved, development would be expected to continue. Another difficulty arises from the undesirability of maintaining two sets of sources, one for the LTSS environment and one for the NLTSS environment. We expect that, ideally, when a real code is moved, the LTSS sources will be frozen and that there will be no looking back. Also, it is not clear at this time whether it would be possible or practical to move a given code system in subphases or whether an entire code system would have to be moved in one fell swoop.

Above all, effective (see "Performance requirements and measurement," p. 20, for a definition of what this means) performance rates will have to be demonstrated before the designers are asked to use an NLTSS-based code system.

There are going to be other sources of difficulty involved in moving the first real production code system. For example, designers will be required to work in two different environments: NLTSS for the converted code system and LTSS for other code systems. Anybody who has had to work with two different text processors or two of anything else realizes the inherent difficulties. This step will require at a minimum the virtual replacement of the LTSS utility-routine environment. Moving files back and forth between environments represents overhead and will be resisted and resented by the designers. For this reason alone, it would be preferable not to make this move until there was some demonstrable benefit to the designer community, such as increased code system functionality, or the opportunity to run on a faster machine or a machine where time is easier to get.

We recommend that the designers not be asked to adapt to a new environment until there is some payoff in it for them. We also need to give serious thought to the question of how to communicate the payoffs to the designers before asking them to adapt.

Specific functionality

The most important source of information generated so far regarding specific functionality is the survey of major NSSD production codes produced by Nancy Alexander and the NSSD-NLTSS Survey Committee, dated January 14, 1983 (Ref. 10). The survey answers must be interpreted in the light of some assumptions that Nancy discusses under "Additional Comments" on the page following the results.

In Table 1 on the following page, we attempt to identify in which phase of transition a given functional requirement is likely to be necessary.

Table 1. Phases of transition and functional requirements.

Functional category	Required functionalities	
	SIMPLE code system phase	Real code system phase
Languages	The ability to compile (CIVIC), load (LDR, BUILD, LIB), and debug (DDT). BCON (not as important as the above). Possible compromises: cross-compilation and loading, static and/or nonsymbolic debugging (caution: this could cut productivity by orders of magnitude).	All SIMPLE capabilities. CFT. COSMOS.
Subroutine libraries[a,b] (notes on p. 29)	A "compatible" BASELIB. BLIB (NSSD). MMLIB (NSSD). ACFLIB (NSSD). PLOTLIB (NSSD). Parts of MATHLIB (this must be made more specific).	All SIMPLE libraries. NLIB. GRAFLIB/GRAFCORE. FORTLIB. STACKLIB. EOSPLIB (NSSD). PSLIB (NSSD). The rest of MATHLIB. See Ref. 10.
Utility routines[c] (notes on p. 29)	File moving between LTSS and NLTSS environments. File services embodied in FILES, COPY, SWITCH, DESTROY, and COMPARE. Text editor, including TMDS capability. User-1 services for text and graphics output. (The text editor and User-1 services could possibly be postponed if the file-moving facility proved adequate for these purposes.)	All SIMPLE routines. File storage and retrieval. Virtually everything in the summary sheets. It is difficult to pin this one down more precisely without surveying every user.
Command language	Log-on service. Ctrl-e queries. The ability to execute processes. The ability to send messages to and from processes.	All SIMPLE services. The ability to monitor multiple jobs.

Notes to Table 1:

[a]Note that Carol Hunter and Jim Kohn are representing NSSD on the NLIB Design Committee and that they are generating documentation which can be referenced here (see Ref. 11).

[b]We also need to address the subject of standard error conventions and the coordination of common blocks (e.g., the use of ZVCACHES in PLOTLIB and FORTLIB).

[c]Additional functionality:
A vital statistics or personality capability (box numbers, TMDS monitor numbers, etc.).
New file management and directory manipulation utilities (see "Archival storage and retrieval," p. 19).

REFERENCES

1. Garry Rodrigue, E. Dick Giroux, and Michael Pratt, "Perspectives on Large-Scale Scientific Computation," *Computer*, October 1980, pp. 65–80.

2. "TMDS and Nuclear Weapon Research," *Energy and Technology Review*, Lawrence Livermore National Laboratory, Rept. UCRL-52000-81-11 (November 1981), p. 16.

3. Stewart A. Brown and Charles F. McMillan, "Proposals for Enhanced Utility of Octopus Under NLTSS," LLNL, undated internal memorandum (1983.)

4. Pierre (Pete) DuBois, "NLTSS — A Status Report," *Tentacle*, August 1983, pp. 7–9.

5. Mike Pratt, "User Requirements Specification for a File Management Utility," LLNL, internal memorandum dated November 18, 1980.

6. Garret Boer and Archival Storage Committee, "Recommendations for Improving Central Storage Service and Performance," LLNL, draft memorandum dated October 5, 1982.

7. J. Fletcher, A. Leibee, J. Minton, and J. Randolph, "Study Committee 1-B Final Report," LLNL, internal memorandum dated January 24, 1979.

8. Tom Slezak, "Unix on Micros and CRAYs or Where Do We Go From Here?" *Tentacle*, September 1983, pp. 22–26.

9. W. P. Crowley, C. P. Hendrickson, and T. E. Rudy, *The SIMPLE Code*, Lawrence Livermore National Laboratory, Rept. UCID-17715 (February 1, 1978).

10. Nancy Alexander and NSSD-NLTSS Survey Committee, "Survey Results," LLNL, internal memorandum dated January 14, 1983.

11. Carol Hunter and Jim Kohn, "Basic External NLIB Requirements," LLNL, internal memorandum dated March 30, 1983.

GS