GCOS-8 SOFTWARE DESIGN SPECIFICATION

GCOS 8 MULTI-SEGMENT ENVIRONMENT

Issued by: A. L. Beard

A. L. Beard, Chairman
Multi-Segment Environment Team

Issue Date: March 31, 1980
Revision Date: March 31, 1980

PREPARED BY:

A. Beard
C. Coflin
F. Little
G. Mann
T. VanVleck
J. Wertz
J. Wilson

APPROVED BY:

G. A. Gillette _____
Director, Multi-Environment Development Center

K. E. Norland _____
Director, Software Systems Engineering

J. R. Roe _____
Director, GCOS Develcpment Center

CONTENTS

Page

Section 3
DESIGN OVERVIEW

CONTENTS (cont)

Section 4
REALIZATION OF GOALS

CONTENTS (cont)

Appencix C
DESIGN EVALUATION

SECTION 1

INTRODUCTION

1.0 Purpose

This document specifies the environment for processes executing in the GCOS 8 Native Mode. The document contains both conceptual and detailed information. Sections 2 through 4 deal with conceptual level information on:

* system goals
* design constraints
* process and module sharing
* interprocess control
* performance considerations

Section 5, to be supplied, will contain detailed information on:

* The nature of the user visible extention to the high level languages required to fully use the Native Mode (but not the exact syntax)
* The format and contents of the files created by the compilers and assemblers
* The calling sequences, conventions and system services used by the object code generated by the compilers
* The JCL and ECL used in the preparation of programs for execution in Native Mode, and the steps required for placing a process in execution

SECTION 2

GOALS AND CONSTRAINTS

1.0 System Goals

From the customer point of view, functionality is the sine qua non of an operating system. We have, therefore, attempted to identify as system goals those functions that will be most important tc both ourselves and our customers. This is not to say that performance has been ignored. Performance represents the primary criterion by which alternate designs for a given functionality are evaluated.

These goals have implications for two different levels of the run-time environment, the macro-structure level and the micro-structure level. The macro-structure of the environment deals with the technology of program management, such as loading, linking, and program library structures. The micro-structure of the run-time environment deals with the technology of program execution, such as calling sequences, scope of reference, address calculation, etc.

1.1 Parc Protection

1.1.1 Migration Support

It must be possible to migrate programs, both user applications and Honeywell procucts, from GCOS-III to the GCOS 8 native run-time environment.

Programs written in high order languages must be able to migrate without source charges.

It is desirable that assembly language programs migrate without source changes.

It is desirable that migration require no job control language changes.

These goals are important for both our customers and ourselves since we both have a large investment in currently

existing programs. The primary implication for the system is that all currently supported higher level language features must have a functional equivalent in GCOS 8.

1.1.2 GCOS-III Slave Mode Accommodation

The run-time environment must support the slave mode execution of GCOS-III executable file formats (Q*, H*, **, etc.). It also must support the following environments:

* TSS slave environment

* DMIV-TP TPR environment

* TDS TPR environment

This objective implies support of a GCOS-III MME interface as defined by some chosen GCOS-III system release.

1.1.3 Performance Relative to GCOS-III

A given application must execute in GCOS 8 native mode with at least 90% of its performance in GCOS-III native mode when utilizing the same resources.

For a given application mix, the total throughput of the system must be at least 90% as good in GCOS 8 native mode as it is in GCOS-III native mode.

These are very ambitious goals, given the system overhead implied in meeting the virtual memory and security goals of the system. It is recognized at the outset that we may be unable to meet the 90% performance figure.

1.2 Ease of Use and Programmer Productivity

1.2.1 Uniform Environment

The macro-structure of the system should have a single personality, for example ECL, that encompasses all methods of user interaction: batch, time sharing, and remote batch.

It is very desirable that the micro-structure of the system have a common run-time environment for both user and system software. This commonality must apply to all module interaction conventions and accesses to services.

Of all the system goals, uniformity has the greatest impact on the long-term software viability. From a human engineering point of view, uniformity reduces cost. From a system

design point of view, its major impact is in simplification and the cost reductions to be gained thereby.

1.2.2 Support High Order Language Conceptual Environments

The run-time environment must support the implementation of high level language features for languages such as COBOL, PL/I, Pascal, Ada, etc. Certain language features will require explicit support ir the run-time environment. For example:

* user visible address values in data space
* automatic space allocation and recursion
* process synchronization
* tasking
* exception processing techniques
* dynamic subprogram irvocation

1.2.3 Support Very Large Applications

The run-time environment must be able to support applications whose procedure space and/or data space may each exceed a segment of 256K words and whose total space requirement does not exceed 1024 segments. The run-time environment should be able to support single structures or arrays that exceed a segment. Such support must not require user preplanning of memory maragement techniques such as program overlays.

The run-time environment should be able to support a large number of files, between 200 and 1000, for each application.

The run-time environment should be able to support a large number of connected terminals, on the order of 10,000 concurrent interactive trarsaction processing users and 5000 simultaneous time sharing users.

1.2.4 Support Multiple Versions of the Same Software Element

The run-time environment should allow multiple versions of both user and system software modules. This capability is necessary to support efficient software development and testing.

1.3 Technological Image

1.3.1 Distributed Systems Architecture Support

The run-time environment must support the Honeywell Distributed System Architecture (DSA).

1.3.2 Virtual Environment

The run-time environment must support a virtual address space that is larger than the real memory. The support technique will be transparent to the user. The run-time environment must not only allow the execution of large programs on smaller real memory but must also provide the effective application of large physical memory.

This goal implies that the run-time environment will use the virtual memory technology of the hardware.

1.3.3 Shared Elements

The run-time environment must support the sharing of unique instances of procedure or data among processes.

1.3.4 System and Applications Security

The run-time environment must guarantee the integrity of all program and data within the system.

The run-time environment must allow user definable access control over units of their applications. The access control mechanism will utilize the hardware segment protection capability.

1.3.5 Dynamic Software Installation

The system will support the addition, deletion, and updating of system software modules without system interruption. It is recognized that the replacement of certain system modules may require system interruption.

1.4 Cost Control

1.4.1 Use Current Hardware

The GCOS 8 run-time environment must utilize the current (NSA) hardware. Although hardware changes are possible, they must be limited to field changeable items. The end

user is indifferent to hardware details as long as his functional and performance needs are met.

1.4.2 Extendible to Future Product Directions

The operating system and run-time environment should insulate user interfaces and system software from evolutionary hardware changes and hardware dependencies.

1.4.3 Protect Honeywell Priced Software

The user manipulation of the system personality must not require change to Honeywell delivered software modules. This objective assumes that the users have valid reasons, such as local accounting conventions, to change the personality of the system. The user must not, however, require access to Honeywell separately priced software products at the source level.

1.5 Goal Summary

The following table summarizes the GCOS 8 system goals and classifies them as to their degree of desirability.

Goal	Rank	Reference
Migrate without HOL source changes	must	1.1.1
Accommodate GCOS-III executable formats	must	1.1.2
Accommodate GCOS-III TSS, TDS, & DMIV-TP	must	1.1.2
Job performance at least 90% of GCOS-III	must	1.1.3
Throughput at least 90% of GCOS-III	must	1.1.3
User visible address values	must	1.2.2
Automatic space allocation and recursion	must	1.2.2
Exception processing	must	1.2.2
Dynamic subprogram invocation	must	1.2.2
Support large procedures	must	1.2.3
Support large data spaces	must	1.2.3
Support Distributed System Architecture	must	1.3.1
Support a virtual environment	must	1.3.2
Support shared elements	must	1.3.3
Provide program and data integrity	must	1.3.4
Provide user access control	must	1.3.4
Use current hardware	must	1.4.1
Uniform micro-structure environment	1	1.2.1
Uniform macro-structure personality	2	1.2.1
Process synchronization	2	1.2.2
Support large number of files	2	1.2.3
Support large number of terminals	2	1.2.3
Support multiple versions of same module	2	1.3.5
Support dynamic software installation	2	1.2.4
Extendible to future product directions	2	1.4.2
Support arrays larger than 256K	3	1.2.3
Protect Honeywell priced software	3	1.4.3
Migrate without assembly source changes	4	1.1.1
Migrate without JCL changes	4	1.1.1
Tasking	4	1.2.2

2.0 Design Constraints

2.1 Business Constraints

The business constraints placed upon the run-time environment are few in number yet very important.

2.1.1 Release with SR2000 (5V)

The multi-segment shared run-time environment must be released for customer use with GCOS 8 Release 2000 (5V). This imposes schedule constraints on the environment, first for a timely definition and second for limiting its content to insure a timely release.

The schedule constraint requires that the run-time environment be consistent with that which exists in GCOS 8, Release 1000 (4VX). It cannot be radically different or the schedule cannot be met.

2.1.2 Minimize Conversion

At this writing, there are interim environments in use and more in development: the ITP environment, the ACOS environment, and the PL-6 environment. Each has sharing mechanisms, calling sequences, and other conventions which differ from one to another.

It is a constraint that modules which execute under one of the interim environments be able to be converted to execute under the new environment with a minimum of change. This is especially true for the modules of ITP. Modules from other environments are of lesser importance.

It is a constraint that domains written according to an interim environment coexist with domains written using the new environment. Adapters may be used as necessary to meet this constraint. It is not required that modules within a domain be mixed - some from an interim environment and some from the new.

2.2 Hardware Architecture Constraints

The multi-segmented run-time environment owes its existence to the NSA hardware. The environment's design, functionality, and performance is totally constrained by the hardware definition. The following sections descuss the more constraining attributes of the NSA hardware.

2.2.1 Access Control

The NSA hardware handles two types of memory space, real and virtual. Only the most privileged modules of the operating system may use real memory addressing. All other procedures must use virtual memory addressing.

The virtual address space of the system is divided into 512 equal length virtual memories called Working Spaces. Working Spaces are accessible to a process via eight Working Space Registers (WSR's). The contents of a WSR cannot be changed by a slave mode instruction. The addressability of a process is thus limited to eight working spaces.

In NSA, a segment is a variable length subdivision of a Working Space. It occupies contiguous virtual memory space and has a homogeneous set of attributes. These attributes differentiate uses of the segments, for example, procedure versus data.

The NSA hardware supports two types of segments, a standard segment with a maximum of one million bytes and a super segment with a maximum size of 64 million bytes.

A segment is controlled by a two-word Segment Descriptor which specifies:

* the particular Working Space containing the segment or a WSR containing the number of to that Working Space,
* the base of the segment relative to a particular Working Space,
* the upper byte address limit in the segment, and
* the valid access rights (read, write, execute) to the segment.

As a means of access control, the hardware requires that all segment descriptors reside in special segments called "Descriptor Segments", recognizable by the hardware, and that these Descriptor Segments, in turn, reside on special pages, called "Housekeeping Pages". The hardware is so designed that Housekeeping Pages can be written to (with normal instructions) only in Privileged Master Mode and can be read (with normal instructions) only in Privileged Master Mode or Master Mode. The address space of a process is thus limited to those segments given to it by the Operating System. Furthermore, since descriptors are not storable in slave data space, they are not usable as address value variables.

All memory addresses have two components, a segment descriptor identification and an offset within the segment. For normal segments, these two components are brought together in the NSA pointer construct. There is no equivalent hardware construct that points into the full extent of a super segment.

2.2.2 Domains

The term "Domain" refers to the particular set of segments that are addressable by a process at any given moment.

A domain consists of a static part and a dynamic part. The static part of a domain is defined by a special Descriptor Segment, the Linkage Segment. There are at most 1024 entries in the Linkage Segment. The dynamic part of the domain is defined by the Parameter Segment and the Data Stack Segment. The Parameter Segment provides for passing arguments into a domain. The Data Stack Segment provides scratch data space.

A domain may include segments in several Working Spaces. During execution, a domain may be augmented by passed parameters or by privileged master mode manipulation of its Linkage Segment.

A process is not restricted to a single domain but will generally execute within several domains. The Linkage, Parameter, and Data Stack Segments are managed by the hardware when changing domains. In using the CLIMB instruction to change domains, however, all of the NSA pointers in data space are invalidated. Furthermore, the CLIMB instruction makes the caller's data stack invisible to the callee. In combination, these constraints complicate exception processing and error recovery.

2.3 High Order Language Constraints

We have a need to support our present-day high order languages such as FORTRAN, COBOL, PL/I, and PL-6, and also to look ahead to the needs of such languages as Pascal and Ada. This need constrains the design of the run-time environment in that it implies many system functions. The following sections discuss those language features that will require explicit support in the run-time environment.

2.3.1 Automatic Space Allocation and Recursion

The dynamic, block structured languages (i.e., all except FORTRAN and COBOL) provice the allocation of automatic data

space whenever a procedure (or block) is entered. The run-time environment, therefore, must contain facilities to grow the data space of a process and must be able to identify the current instance of the data space that is to be referenced by the procedure. The automatic space allocation technology is necessary to support the general recursion facility offered by these languages.

2.3.2 Dynamic Space Allocation by the User

In addition to the automatic space allocation feature, PL/I, Pascal, and Ada allow the user to dynamically allocate space. The International Organization for Standardization (ISO) is considering the addition of this facility to FORTRAN. A variation of dynamic allocation is implied by the COBOL CALL/CANCEL facility. This language feature implies the same type of run-time facility for the growth of process data space as is implied by the automatic space allocation facilities.

2.3.3 User Visible Address Values

PL/I, PL-6, Pascal, and Ada allow programs to declare variables containing address values. These variables may appear within data structures and, all languages considered, may point anywhere within the static, automatic, or dynamic data space of the program. When created, these address values should remain valid throughout the life of the program. Since any datum may be used as an argument to another procedure, the value of a address variable should remain useful across some depth of procedural calls.

Since all of the high order languages support or plan to support bit aligned data, address values must be able to resolve storage to the bit level.

2.3.4 Exception Processing

All of the high order languages except Pascal have specified some facility for handling error conditions or exception procedures. There is no consistency in these facilities from language to language. The run-time environment must therefore provide an exception handling technology that is sufficiently robust to support the full spectrum of language specifications.

2.3.5 Data Space Initialization

All of the high order languages allow the user to specify initial values for some classes of data. The run-time environment must contain the controls in both its micro-structure and its macro-structure to allow these initial values to be realized at execution time.

2.3.6 Process Synchronization

Several of the high orcer languages support semaphore or signal constructs that may be used to communicate between separately scheduled processes. Typically, these features are used to synchronize two or more processes. In supporting these features, the operating system may require special help from the structures of the run-time environment.

2.3.7 Tasking

In addition to process synchronization, some languages provide the ability to initiate the separate scheduling of a separate process. As in synchronization, the operating system may require special help from the run-time environment.

2.3.8 Subprogram Invocation

All of the languages provide for the invocation of separately compiled programs. Current structured programming technology encourages the use of this facility. Therefore, the calling sequence technology of the run-time environment will be an important determinant of system performance. The technology must also support the various language specifications for passing arguments by reference or by value. The ANS COBOL specification expands the problem in two ways. Its CALL/CANCEL facility allows the dynamic association, invocation, and disassociation of subprograms whose names are supplied at execution time. This facility will require special run-time environment techniques in both program invocation and program packaging. The COBOL SORT/MERGE facility allows the user to establish procedures within his program that are to be used as co-routines by the system software that supports the sort or merge. The run-time environment must solve the program packaging problem posed by such co-routines and must provide an extremely efficent invocation technology for them.

SECTION 3

DESIGN OVERVIEW

1.0 System Concepts

1.1 Conceptual Model

The operating system is described in the accompanying GCOS 8 ARCHITECTURE document as a layered construct at the center of which resides an inviolate system kernel. That kernel provides the hardware dependent functions and those housekeeping functions that require privileged master mode execution. System shared software is closely associated with the kernel. This software provides the common service functions that are used by all users, regardless of their interface to the system. The outermost layer of the system provides the end-user interfaces that define the personalities of the system.

The system is also planned to be naturally adaptable to the Distributed Systems Architecture. Support services such as session control and workstation management will be supported in the system shared software.

1.2 System Organization

Both the construction and utilization of the system are organized around the concept of segmentation. In terms of construction, segments may be considered singly or may be organized into domains. These single segments and domains are, in turn, organized into Working Spaces for the sake of execution.

Utilization of the system is organized around the concept of "process". A process is a triplet composed of an execution stream, its associated data, and the processor that is doing the execution. The execution stream may involve several procedure segments and/or domains and/or Working Spaces in succession. It may also involve many data segments in disparate domains and/or Working Spaces.

The organization of the run-time environment has two different levels, the macro-structure level and the micro-structure level. The macro-structure of the environment deals with the technology of program management, such as loading, linking, and program library structures. The micro-structure of the run-time environment deals with the technology of program execution, such as calling sequences, scope of reference, address calculation, etc.

2.0 System Macro-structure

2.1 The Virtual Environment

The virtual environment is defined by the segments available to the system. The segments are organized into Working Spaces. The virtual environment available to a process is limited to those Working Spaces that are addressable through the Working Space Registers (WSR's). The virtual memory local to the process itself, that is, its segments and domains, are rooted in a single Working Space. That Working Space is accessed via WSR7. The other WSR's are loaded to provide the process access to system level and shared domains and segments.

All Working Spaces have the same general structure, although all types of segments dc not exist in every Working Space. This consistency of structure across Working Spaces permits easy access to data that is canonically located within the Working Space.

2.2 The Sharing Mechanism

Resource sharing is an important objective of GCOS 8. However, this sharing of resources must be balanced with another objective, security. The criteria for security are that, without proper authority, no user should be able to:

* retrieve another user's data or programs

* manipulate another user's data or programs

* deny the resources of the system to another user

These criteria imply that resource sharing, while desirable, must be tightly controlled. The system must be protected from the external users and the users from each other.

This isolation is accomplished at four levels:

1. Working Space Level - a Working Space is addressable only if the Working Space Number is loaded into one of the Working Space Registers for the process.

2. Page Level - to reference a page, it must be mapped into the page table and the reference must be consistent with the housekeeping and write protect flags in the Page Table Word.

3. Domain Level - to reference a datum, a segment descriptor for the segment containing the datum must be accessable in the dcmain.

4. Segment Level - the data reference must be within the segment's bounds and must be consistent with its type field (data, descriptor, or entry) and permission flags (read, write, execute, etc.).

When two or more processes share a Working Space, sharing takes three forms, domain instance sharing, domain pattern sharing, and segment sharing.

2.2.1 Domain Instance Sharing

The first form of sharing is the sharing of a unique instance of a domain. There is one Linkage Segment for the shared domain and all processes CLIMB to the domain via identical entry descriptors to that single Linkage Segment.

A consequence of domain instance sharing is that all static segments of the domain are shared. There are no process local segments accessible to the shared domain other than those passed as parameters of the CLIMB.

2.2.2 Domain Pattern Sharing

The second form of sharing is the sharing of a pattern or template for a domain. A skeleton Linkage Segment is used as a pattern to create multiple occurrences of the domain. Each occurrence of the domain is created by allocating one or more data segments in the invoking domain and inserting them into the skeleton Linkage Segment. The resulting Linkage Segment, i.e., domain, will embrace shared procedure segments local to the new domain and data segments in either the caller's Working Space or in the shared Working Space or both.

This type of sharing is useful when the domain must include both shared and process local segments. The shared domain pattern includes the shared segments, but each occurrence of the Linkage Segment is given separate instances of the local segments. Since the one pattern is always used to construct the domain occurrences, all the Linkage Segment occurrences have the same layout or definition.

2.2.3 Segment Sharing

The third type of sharing is segment sharing. In this case, individual segments are shared among multiple domains. This type of sharing has a number of restrictions which, when met, allow very efficient operation.

The restrictions which apply to shared segments are:

1. If the segment is a procedure segment, it may access descriptors in the comain's Linkage Segment or Parameter Segment only when they are in fixed, canonical locations or when pointers (NSA pointers) are passed as arguments of a call.

2. Procedure segments must be pure.

3. Data segments, if impure, must be gated by means of a monitor, i.e., access to them must be through a monitor procedure.

2.3 Working Space Packaging

2.3.1 Overview

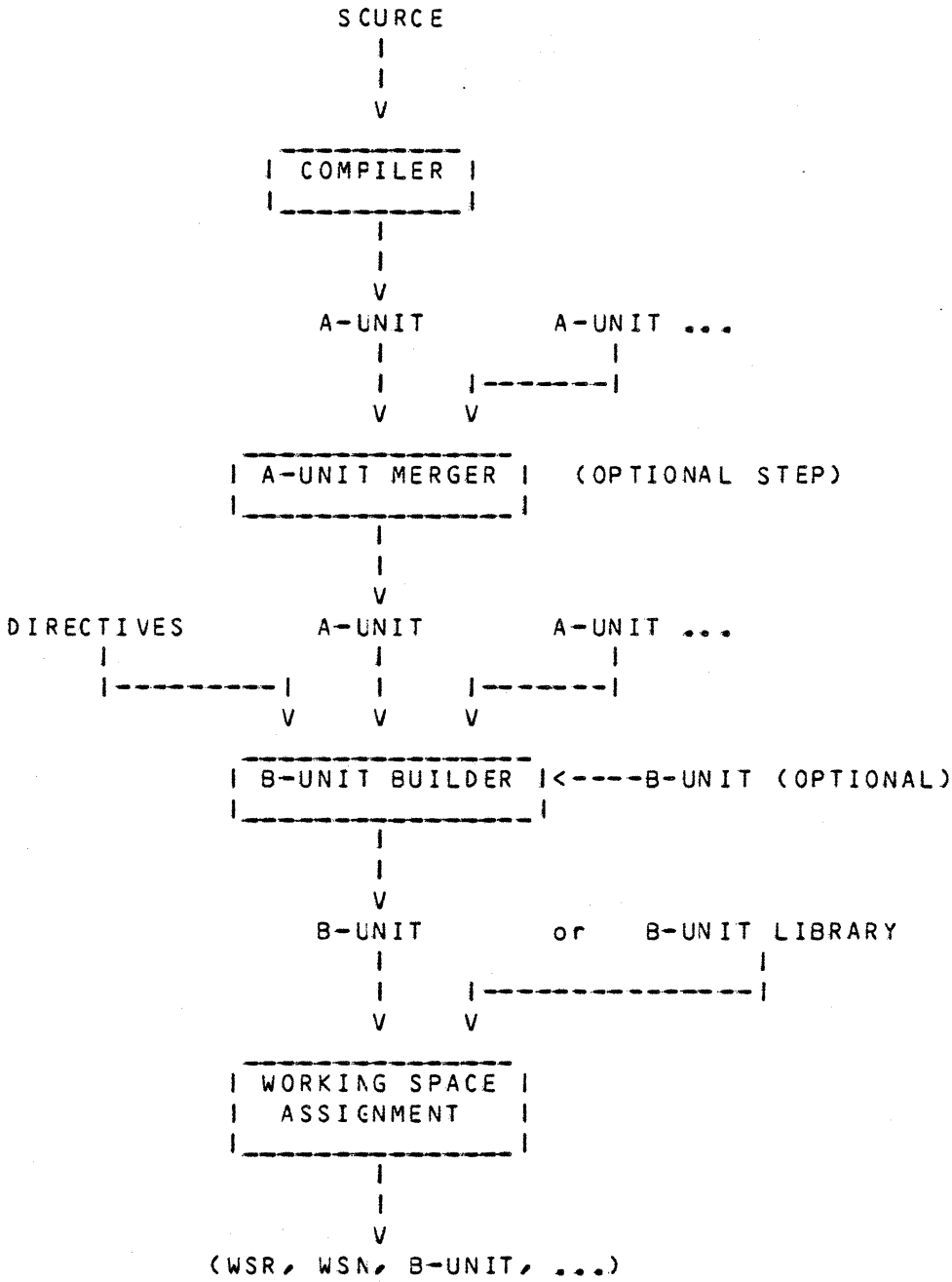
The following figure depicts the steps required to prepare a program for execution, beginning with compiling the program and ending with the mapping of the program into a Working Space.

A compiler or assembler produces an A-unit from the source program. The A-unit contains the initial segment contents, both code and data.

If desired, multiple A-units may be combined into one A-unit. The result of this merging is that segments with compatible attributes are combined, thereby reducing the total number of segments required.

Next, one or more A-units are input to the B-unit Builder which combines them into domains, as specified by the directives, to produce a B-unit. The B-unit is a file containing a Working Space image of the domains and their segments. Optionally, an existing B-unit may have one or more A-units added, deleted, or replaced.

The Working Space Assignment function assigns a Working Space Number and a Working Space Register to the B-unit. Execution commences when the process structure is added to the Working Space by the Process Initiator and the root domain of the process is dispatched.



2.3.2 Packaging Elements

2.3.2.1 A-unit

An A-unit is a file which contains the object representation of an independently compiled or assembled program unit (e.g., a PL/I external procedure or a COBOL program). The creation of an A-unit is not a privileged operation. All

compilers and assemblers produce an A-unit as output. An A-unit contains the follcwing types of information :

* A-unit icentificatior
* segment cefinitions and references (SEGDEF/SEGREF)
* domain definitions and references (ENTDEF/ENTREF)
* symbol definitions and references (SYMDEF/SYMREF)
* object text
* debug schema
* relccaticn informaticn
* resource requirements

Before a prcgram represented by an A-unit can be executed, it must be combined to form one or more B-units by the B-unit Builder.

2.3.2.2 B-unit

A B-unit is a file which contains a representation of one or more dcmains, including procedure, data, and Linkage Segments for each domain. B-units are produced by the B-unit Builder from one or more A-units. The B-unit Builder is able to "update" an existing B-unit by adding or replacing A-units. A B-unit contains the following types of information :

* B-unit identificatior
* skeletal page table (describes virtual space assignment)
* one cr mcre domains
* Linkage, procedure, and data segments for each domain
* Domain Directory of all domains (ENTDEF's)
* Global Segment Directory for all segments known externally to the B-unit
* directory of unresolved segment references (SEGREF)
* directory of unresolved domain references (ENTREF)

Note that a B-unit does not contain any process structure (e.g., hardware stack segments, SSA segments, etc.). All

references to segments and domains outside the B-unit are left unresolved.

2.3.2.3 Libraries

It is necessary at times to reference groups of files as a unit. For example, a group of B-units may be assigned to a Working Space. This is accomplished by referencing a library containing those B-units. A library is implemented as a directory in the File System. This directory contains only files, no subordinate directories, and all these files are of the same type (e.g., B-units). Since the files are of the same type and have common attributes such as control interval size, access to these files can be optimized.

2.3.3 Compiling

The compilation process employed for the GCOS 8 environment is the conventional one in which source programs in the form of text files are processed to produce object modules in the form of A-units, and, optionally, a report of the compilation process in the form of a listing. Such A-units usually require the support of runtime libraries for their execution and may require other user-supplied A-units for their execution.

A typical A-unit will contain two or more segments (at least one instruction segment and one data segment). However, some language constructs or implementation techniques may produce large numbers of segments.

2.3.4 A-unit Merging

The merging of A-units is combining the segments of two or more A-units into fewer total segments. The segments with compatible attributes are combined and relocation is performed on the segment references. Thus all of the procedure segments, one for each A-unit, might be combined to form only one procedure segment. The output of the A-unit Merger is a new A-unit that contains the segments of all of the input A-units.

2.3.5 B-unit Builder

The B-unit Builder produces a B-unit from one or more A-units and a set of directives that describe how these A-units are to be combined into domains. The B-unit Builder creates a Linkage Segment for each domain and assigns virtual space to each segment of the domain starting at a conven-

tional base address. Domain and segment references are resolved where possible. A page table that describes the assigned virtual space is also created.

The B-unit that is produced includes a Domain Directory, a Global Segment Directory, and a directory of unresolved segment references. The Domain Directory contains an entry describing each domain ir the B-unit, while the Global Segment Directory contains an entry describing each segment in the B-unit which may be referenced from outside the B-unit.

Sufficient information is kept in the B-unit to permit the addition, deletion, or replacement of one or more A-units in an existing B-unit.

No shared libraries are referenced in order to create the B-unit. All references outside the B-unit are left unresolved. References to external domains result in a dynamic linking descriptor that references the name of the domain. Segment references result in segment descriptors with the "missing segment" attribute. These dynamic references will be resolved by the Dynamic Linker at process initiation time.

2.3.6 Working Space Structure

All Working Spaces have the same general structure. At the beginning of each Working Space, offset zero, is a descriptor segment which serves as a directory to the Working Space. This Working Space Unique System Header (WSUSH) has the same canonical definition for each Working Space, regardless of the function for which the Working Space is employed. For a given Working Space, not all entries of the WSUSH are valid (e.g., the entry for the SSA Segment is not valid for a shared Working Space). Invalid entries contain null descriptors.

The segments located via the WSUSH fall into two categories: those required for all Working Spaces regardless of their function and those required only for Working Spaces that instantiate a processes.

2.3.6.1 Segments Required in All Working Spaces

All Working Spaces require at least the following segments:

* the Domain Directory, a table that defines every domain in the Working Space.

* the Global Segment Directory, a table that defines every global segment (i.e., every segment known externally) in the Working Space.

* a directory of all the dynamically allocated segments.

* the Page Table, a variable length segment containing the page table entries for the Working Space.

* the PAT segment, a variable length segment used to contain the Peripheral Allocation Tables (PATs) for the Working Space.

* the DCW buffer, a segment used for DCW list storage for paging I/O, process swapping I/O, SYSOUT I/O, etc.

2.3.6.2 Segments Required in Process Working Spaces

In addition to the segments required in all Working Spaces, those Working Spaces usec to contain processes also require at least the following segments:

* the Exception Procedure Entry Descriptor Segment (EPEDS), a descriptor segment containing the entry descriptors to the exception handling procedures for the defined exception corditions.

* the User's Linkage Segment Descriptor Segment, a variable length descriptor segment containing the Linkage Segments for all user domains in the process.

* the Safestore Stack Segment, a variable length segment used to store registers when changing domains.

* the Argument/Parameter Stack Segment, a variable length segment used to pass arguments between domains.

* a segment used by the Dispatcher.

* the SYSOUT segment, used to collect the system output records. The size, content, and location of this segment vary with the number of SYSOUT lines.

* the SSA Data Segment, a segment containing control information equivalent to the GCOS-III control information contained in the Slave Service Area (SSA).

* the SPA Data Segment, a segment containing control information equivalent to some of that contained in the GCOS-III Slave Prefix Area (SPA).

* the Process Control Block (PCB), a segment containing information necessary to control the process.

2.3.7 Assigning a B-unit to a Working Space

A B-unit may either be placed in execution, that is, become a process, or be a shared B-unit, that is, be referenced by or executed by many processes. In either case, the B-unit must be assigned to a Working Space. The function of Working Space Assignment is not to load B-units into the virtual memory of the Working Space, but to associate the B-units with the Working Space. This is accomplished by constructing a Domain Directory and a Global Segment Directory in the virtual memory of the Working Space. These directories completely define all domains and global segments contained in the B-unit or B-unit Library. These directories are searched by the Dynamic Linker when attempting to resolve a reference to a domain or segment. At the first reference to a domain the B-unit containing the desired domain or segment will be loaded.

The input to the Working Space Assignment function is either a single B-unit or a B-urit Library that is to be assigned a Working Space. The resource requirement information is read from the B-units and is used to create a backing store file. The Domain Directory and Global Segment Directory are read from each of the B-units. These directories are combined and are located canonically in the Working Space.

After completion of the Working Space Assignment step, a skeletal page table, Domain Directory, and Global Segment Directory exist in virtual space and a backing store file will have been created for the Working Space. An available Working Space Number will have been assigned. The assignment of a Working Space Register for the B-unit will depend on whether the Working Space is to be shared or is to become the root of a process.

2.3.8 Working Space Register Usage

The fundamental sharing mechanism in GCOS 8 is the sharing of domains and segments in shared Working Spaces. By mapping Working Spaces into the same virtual address space of a set of processes, the contents of the Working Spaces may be shared among the processes.

Within GCOS 8, WSR7 is reserved for all process local information. The other WSR's are used for shared software and data. The smaller the WSR number, the more global the sharing. The provisional assignment of WSR's and sharing is as follows:

WSR_Number	Usage
0	Operating System Hard Core
1	Operating System Hard Core
2	Operating System Slave Mode
3	Priced Software Products
4	Installation (Site) Specific Software
5	User Shared Software
6	Workstation Local
7	Process Local

The mapping is performed by the WSR contents. A Working Space is shared if two or more processes have its Working Space Number in the same WSR. When a Working Space is shared, it must be shared by using the same WSR in all the sharing processes. This restriction is due to the fact that references to the WSR appear in the Working Space itself. Each segment descriptor contains a value for the WSR containing the segment it describes.

Furthermore, once two processes have established the sharing of a Working Space in some given WSR, all of the more global WSR's must have matching values for the two processes.

The Working Spaces referenced by WSR0 through WSR4 are shared by all processes in the system. WSR5 is reserved for customer controlled sharing. WSR6 contains the same value for all processes of a workstation. The content of WSR7 is unique to each single process.

2.4 Process_Execution

2.4.1 Process_Initiation

The B-unit destined to become a process, having been assigned to a Working Space, now only requires the addition of the process structure to be executable. The Process Initiator assigns a process number, builds the process structure (e.g., hardware stacks, SSA segment, process control block, etc.), and loads the Working Space Registers for the process.

The B-unit itself has still not been loaded in virtual memory. Only the "definition" of the B-unit, in terms of the names of its domains and global segments and its process structure have been loaded.

Finally, the Process Initiator executes a CLIMB instruction to the user entry point. Since the B-unit containing the user's domain has not yet been loaded, this CLIMB generates a dynamic linking fault. The Dynamic Linker resolves the

reference and the process begins executing in its root domain.

2.4.2 Dynamic Linking

The Dynamic Linker is invoked to resolve references to both segments and domains. This involves utilizing the search rules that govern the orcer in which the Working Spaces are searched, lccating the desired object, and replacing the unresolved reference with the appropriate entry or segment descriptcr.

Linking to shared domains occurs dynamically while the process is in execution, while linking to shared segments occurs at the time when the B-unit containing the reference is loaded into virtual memcry. References to shared segments are resolved by locating the segment and storing the descriptor of the segment in the referencing domain (i.e., Linkage Segment).

2.4.2.1 Search Rules

Whenever a dynamic reference to a segment or domain occurs, a search must be conducted in an orderly manner through the virtual space addressable by the executing process; that is, through the Working Spaces loaded behind the WSR's for that process. The search begins with the Working Space containing the instruction segment of the executing domain and proceeds sequentially through more global (decreasing) values of Working Space Register number. For example, if while executing a domain whose procedure segment is behind WSR5 and a dynamic linking fault occurs, then the search for the referenced domain begins with WSR5 and continues in sequence through WSR4, WSR3, WSR2, WSR1, and WSR0 until the desired domain is found.

At times it is desired to reference a domain at a lesser scope of sharability, that is, a domain behind a higher value of WSR. This functicnality is useful in the support of exception processing, user exit procedures, etc. This "outward" reference will be allowed only when explicitly declared on the reference. In this case, the dynamic linking descriptor contains a field which specifies the degree to which the outward reference is permitted.

2.4.2.2 Dynamic Linking to Domains

Each Working Space contains a Domain Directory that is located canonically via the WSUSH and describes the domains

of the Working Space. Each entry in the Domain Directory contains at least the following information:

* domain name

* name of B-unit cotairing the domain

* entry descriptor to comain (valid only if the B-unit has been activated)

* domain type (unshared, shared domain, shared domain occurrence, etc.)

* count of outstanding references to the domain

When a dynamic linking fault is generated by the execution of a CLIMB instruction through a dynamic linking descriptor, the Dynamic Linker is invoked to resolve the domain reference. If the domain is not found, the Dynamic Linker returns an error status and exception processing commences. If the name is found ard the B-unit containing the domain has not been loaded, then the B-unit is activated. The reference to the domain is then resolved depending upon the WSR's behind which the invoked and invoking domains are found and upcn the domain types.

If the referenced domain uses unique domain instance sharing and the referenced domain has been found behind a more global WSR than the referencing domain, then the dynamic linking descriptor is replaced with the actual entry descriptor to the shared domain and the CLIMB is re-executed.

If the referenced domain uses unique domain instance sharing but has been found behind a less global WSR than the referencing domain, then the CLIMB is completed without replacing the dynamic linking descriptor in the referencing domain.

If the referenced domain uses domain pattern sharing, the prototype Linkage Segment is copied into the caller's space. Any local segments are created dynamically and initialized. Then the original dynamic linking descriptor in the calling domain is replaced by an entry descriptor to the newly created Linkage Segment and the CLIMB is re-executed.

2.4.2.3 Dynamic Linking to Segments

Each Working Space contains a Global Segment Directory that is located canonically via the WSUSH. This directory describes the segments of that Working Space which are externally visible (i.e., shared among B-units). Each entry in the directory contains at least the following information :

* segment name

* name of B-unit containing the segment

* segment descriptor (valid only if the B-unit has been activated)

* count of outstanding references to the segment

If, when loaded into virtual memory, a B-unit contains any unresolved references to segments, then the Dynamic Linker is invoked to resolve these references before the B-unit is executed. The Dynamic Linker employs the search rules to determine the Working Spaces to be searched and then searches the associated Global Segment Directories for the desired segment name. If the name is not found behind a more global Working Space Register, a descriptor with the "missing segment" flag set is returned. If the name is found and the B-unit containing the segment has not been loaded, then the B-unit is activated. Finally, the descriptor framing the desired segment is returned.

2.4.3 B-unit Activation

A B-unit is activated in response to a call from the Dynamic Linker when attempting to resolve a reference to a segment or domain. The referenced B-unit is assigned an origin or base for data segments and another for descriptor segments. All of the descriptor segments for the B-unit are then loaded in virtual memory.

All of the segment descriptors and entry descriptors in the B-unit were initialized with a value for the Working Space Register (WSR) when the B-unit was created. If that value for WSR is not the same as that assigned by the Working Space Assignment function, then the WSR values in the descriptors must be adjusted to the correct value. If the base virtual addresses for both data and descriptors assigned by the B-unit Activator do not agree with those assigned by the B-unit Builder, then the base virtual addresses in the descriptors must also be adjusted.

The page table for the Working Space is updated to reflect the addition of the pages for the B-unit and the backing store file may be initialized at this time. The real memory working set is also adjusted to reflect the addition of the B-unit to the Working Space. The Domain Directory and the Global Segment Directory are updated to reflect the actual virtual memory address of the domain and segment (i.e., entry and segment descriptors) in the B-unit.

Finally, the B-unit Activator must determine whether this B-unit itself has any urresolved segment references. This is accomplished by accessing the table of unresolved segment references in the B-unit. For each unresolved reference, the B-unit Activator calls the Dynamic Linker. This, in turn, may cause other B-units, the ones containing the referenced segments, to be activated. When this process is complete, the B-unit activation has been finishec.

Note that references frcm the B-unit to segments have now been resclved. However, references to other domains outside the B-unit have not. Dcmain references still exist in the form of dynamic linking cescriptors.

2.5 Inter-Process Communicatisn and Synchronization

The Process Synchronization facility of GCOS 8 exists to perform two tasks:

* maintain the integrity of shared data, and

* synchronize the execution of parallel process.

A more specific discussion of these concepts is to be supplied.

3.0 System Micro-structure

The micro-structure of the run-time environment consists of the conventions for:

* intra-domain calling sequences
* inter-domain calling sequences
* stack handling
* register allocation
* exception processing
* interrupt handling
* condition handling
* inter-process synchronization
* operators
* debugging aids
* segment structure and binding

3.1 Standard Segments

Each domain contains a number of standard segments. They are:

* Linkage Segment
* Parameter Segment
* Argument Stack Segment
* Software Stack Segment
* Procedure Segment(s)
* Data Segment(s)
* Operator Segment

3.2 Software Stack Conventions

Each domain has a Software Stack Segment for argument passing and subroutine linkage within the domain. The Software Stack Segment may be a static part of the domain or it may be dynamically obtained from the Data Stack. If it is in the

Data Stack, the entire amount required by the domain is allocated upon entry.

The descriptor framing the complete Software Stack segment is saved in location 0 of the Argument Stack and in a conventional ODR. The associated pointer register always points to the base of the current stack frame.

There are two kinds of stack frames in the Software Stack -- a root frame and a basic frame.

3.2.1 Root_Frame

There is one root frame in the Software Stack and it is always the first frame. It is created on domain entry. The root frame contains the following information:

* a fault recursion count
* a pointer to the exception processing array
* the base of the current stack frame
* the total size of the stack
* the location of all default enabled conditions, and
* the location of the next available stack frame.

The root frame is updated when each internal call is made, i.e., when a basic frame is created or released.

3.2.2 Basic_Frame

There are many basic frames in the Software Stack. A basic frame is created when a subroutine is called, e.g., external procedures, ON CONDITION handlers.

A basic frame contains a fixed area for control information and a variable length area for parameter passing and the subroutine's (block's) automatic storage.

The information consists of:

* register safe store (optional)
* control information
* pointers to input and output parameters
* pointers to argument descriptions (optional), and

* automatic storage.

It is important to note that the parameters passed in the stack are the addresses of the data items. In GCOS 8, these addresses are NSA pointers having a 24-bit bit address and a 12-bit segment identification.

3.3 Operators

Included in every domain is a segment reference to a shared procedure segment containing operators. Operators are short procedure sequences which perform some support service to the compiled procedure. Among the operators are code sequences to handle intra-domain procedure calls (between segments of the domain) and exits, various arithmetic functions, operating system call adapters (PMME adapters) and inter-domain calls and returns.

Operators are invoked by an inter-segment (cross-segment) transfer to the correct entry point in the operator segment.

The subroutine linkage operators perform all the stack handling and environment preparation required during a subroutine call. The preparation of arguments is done before the operator is invoked.

3.4 Intra-domain Calling Sequence

The calling sequence used within a domain establishes conventions for how the parameters are passed, how the Software Stack is handled, how the return linkage is handled and how the callee's environment is created.

3.4.1 Subroutine Linkage

The actions that are required for subroutine invocation are divided between the calling procedure and the interface operator. The calling procedure prepares the arguments and argument descriptions while the interface operator handles the stack, does any register saving and creates the return linkage.

3.4.2 Parameter Passing

Parameters are passed from the caller to the callee by passing a list of addresses of the parameters plus (optionally) the addresses of their argument descriptions.

Since the addresses are NSA pointers, the descriptors of the segments which form the comain must be in either the Linkage Segment, the Parameter Segment or the Argument Stack. Subordinate descriptor segments cannot be used.

Note that no parameters or addresses are passed in registers. This insulates one subroutine from the ODR and/or register conventicns of another.

3.5 Program Segment Structure

The output of a compiler is an A-unit containing procedure and/or data segments. A procedure segment consists of generated code, a pointer area, and one or more entry areas. The generated code is all IC relative, i.e., it is floating code.

Except for entry points, there are no references to a procedure segment from outside the segment. Constants are packaged within the procedure segment, thus the minimum permissions for the segment are Read and Execute.

3.5.1 Pointer Area

The pointer area is an area containing all the NSA pointers needed by the procedure for references to other segments as shown in Figure 3(a). The procedure loads ODR's (via LDPi instructions) from this area when necessary. All references to the pointer area are IC relative.

3.5.2 Entry Point Structure

Associated with each entry point to a procedure segment is data which defines:

* The ASCII name of the entry point
* The number of parameters expected
* The language and version which created the procedure
* The amount of automatic (stack) storage necessary
* The location of the executable procedure, the pointer area and debugging irformation (debug schema).

This data is tentatively located at negative offsets relative to the entry point.

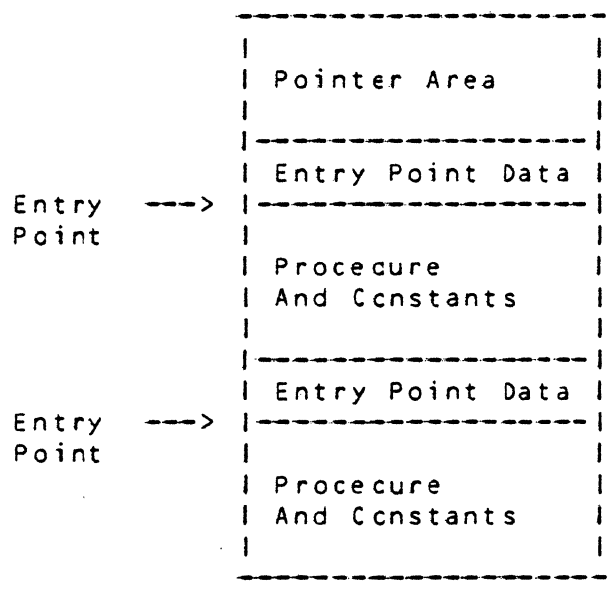


Figure 3(a) Procedure Segment Layout

3.5.3 Procedure Segment Merging

Binding a procedure segment into a domain involves resolving the inter-segment references contained in the pointer area. This will cause the SEGID and the 24-bit address fields of each pointer to be adjusted as the Linkage Segment of the domain is established. Only the pointers which refer to the Linkage Segment are adjusted. Those which refer to the Parameter Segment and Argument Stack do not require adjustment (relocation).

Multiple procedure segments may be merged into one segment during the binding process. This is possible when their combined size is less than 256K and their attributes (execute, read) are identical. Since the generated code is floating code, two procedure segments can be combined into one by concatenating them and adjusting the inter-segment references of all segments in the domain.

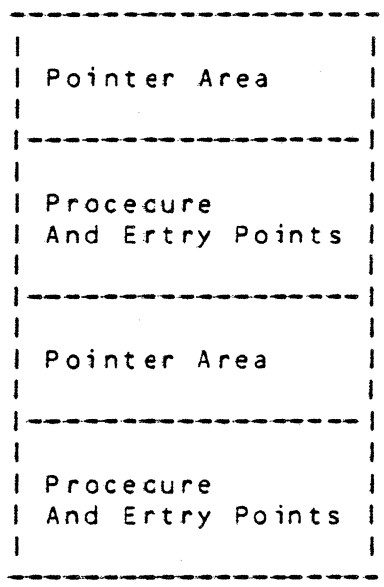


Figure 3(b) Merged Procedure Segments

3.5.4 Data Segment Merging

Multiple data segments may be merged into one segment during the binding phase. This is possible when their combined size is less than 256K and their attributes (read, write, cache-bypass) are identical.

References tc the segmerts which have been merged must be adjusted by relocating the NSA pointers which form the references. Similarly, references from one data segment to another via NSA pointers (which arise from pointer data types) must be adjusted, both ir their SEGID field and their 24-bit address field.

3.6 Exception Processing

Exceptien processing includes the handling of:

* ON CONDITIONS
* faults
* interrupts

Each is handled by a cordition handler unique to the event. ON CONDITION events may be detected synchronously during

normal processing or they may be detected asynchronously by a hardware fault. Another asynchronous event which is handled the same way is a "software interrupt", i.e., an interrupt from one process to another.

The things which are used in handling exception conditions are:

* The Exception Processing Pointer Array (EPPA)

* The ON CONDITION handlers.

* The Exception Processing Entry Descriptors (EPEDS)

3.6.1 Exception Processing Pointer Array

Associated with every domain is an array which contains pointers to the asynchronous event processing routines for the domain. This array is located by a pointer in location 0 of the stack segment, which is in turn located by a descriptor in location 0 of the Argument Stack.

The EPPA may be in its own segment or may be part of a larger segment. The EPPA contains NSA pointers to the procedures which handle

* The hardware faults (overflow, lockup, etc.)

* Software interrupts

* GELOOP detection

* Wrapup

* Restart.

3.6.2 Exception Processing Entry Descriptors

To be supplied.

3.6.3 ON CONDITION Handlers

To be supplied

3.6.4 Exception Processing Flow

To be supplied.

SECTION 4

REALIZATION OF GOALS

1.0 Introduction

This section describes how the proposed environment does or does not meet the goals that were stated in Section 2. The summary table from Section 2 is reprinted with a column which indicates whether the proposed design will meet the goal, whether it will not meet the goal, or whether its response to the goal still needs to be determined. In those cases where a simple answer will not suffice, the column contains a reference to a succeeding paragraph in this section.

2.0 Goal_Realization_Summary

Goal	Rank	Response
Migrate without HOL source changes	must	yes
Accommodate GCOS-III executable formats	must	yes
Accommodate GCOS-III TSS, TDS, & DMIV-TP	must	yes
Job performance at least 90% of GCOS-III	must	3.0
Throughput at least 90% of GCOS-III	must	3.0
User visible address values	must	yes
Automatic space allocation and recursion	must	yes
Exception processing	must	TBD
Dynamic subprogram invccation	must	TBD
Support large procedures	must	yes
Support large data spaces	must	TBD
Support Distributed System Architecture	must	yes
Support a virtual environment	must	yes
Support shared elements	must	yes
Provide program and data integrity	must	yes
Provide user access cortrol	must	yes
Use current hardware	must	yes
Uniform micro-structure environment	1	yes
Uniform macro-structure personality	2	yes
Process synchronizatior	2	yes
Support large number of files	2	yes
Support large number of terminals	2	yes
Support multiple versicns of same module	2	yes
Support dynamic software installation	2	yes
Extendible to future product directions	2	yes
Support arrays larger than 256K	3	no
Protect Honeywell priced software	3	TBD
Migrate without assembly source changes	4	4.0
Migrate without JCL changes	4	4.0
Tasking	4	TBD

3.0 System Performance Estimates

Two types of performance analysis were done. The first analyzed programs executing in an existing multi-segment environment using four performance case studies. The programs were analyzed to determine their instruction mix and then, from the mixes, the perfcrmance of the programs relative to their execution on the ursegmented GCOS-III was estimated.

In the second analysis, the object code of two programs compiled for GCOS-III was modified for the multi-segment environment and analyzed relative to the original versions.

It is important to note that many factors in addition to the execution environment affect the performance of the system. The analyses presented in this section do not predict the overall GCOS 8 performance relative to GCOS-III. Rather the numbers state that for a given number of instructions executed, the GCOS 8 performance will be b times the GCOS-III performance. Since b is less than one, using the multi-segment capability of the NSA hardware in the GCOS 8 environment effectively ce-rates the CPU. Other factors not included in this analysis such as the differences in the supporting run-time subroutines, the operating system services, etc., will significantly affect the total performance of GCOS 8.

3.1 Performance Case Studies

References :

1) Ireland, R.J. and O'Laughlin, J.T.,
"Virtual Unit Instructions, Times, and Counts",
Analysis Note -- 182,
February 14, 1980.

2) Brown, F.M.,
Vue-graph tables on NSA instructions dated
January 28, 1980.

3) Ireland, R.J., private communications on NSA timing,
February 18, 1980.

4) Krasny, L.,
"Virtual Unit Instructions on CP-6",
March 11, 1980.

The interesting combinations of hardware and operating systems are presented in the following table, using GCOS III performance on the 6680 without the NSA option as a baseline.

	GCOS III	GCOS 8 Accommodation	GCOS 8 Native
6680	X	aX	bX

The coefficients "a" and "b" represent the performance factors. Due to the pipelire structure of ADP, it is impractical to derive the ADP coefficients without simulation or measurement. Therefore, this study only attempts to derive values for the 6680 coefficients "a" and "b".

A number of case studies are presented, some representing static analyses of various programs and some representing actual measurements. All of the analyses calculate figures for instruction mix, particularly of NSA instructions, and based on the instructior mix and the timing of the various instructions, derive the coefficient "b". Coefficient "a" is determined from an actual measurement.

The calculations of the two coefficients are based on two assumptions:

1. The non-NSA instructions in the GCOS 8 environment take an average of 1.735 microseconds (Ref. 1).

2. Instructions in the GCOS III environment take an average of 1.644 microseconds (Ref. 1,3).

3.1.1 Summary

The following table summarizes the results of the various case studies.

	6680 Performance Coefficient
CASE STUDY 1	
Accommodation Mode	.93
NSA instruction use in native mode is in same proportion as measured in SR1000 master mode.	.88
CASE STUDY 2	
CP-6 Scrt Command Executive	.827
CP-6 Sort Tournament Driver	.928
CASE STUDY 3	
SR1000 Global Data Management	.86
CASE STUDY 4	
CP-6 Measurements	.94

3.1.2 Conclusions

Although case studies 2 and 3 are based on a static analysis rather than actual measurements, the results correlate quite closely with the measured results in case studies 1 and 4.

Two factors are seen tc affect performance for the 6680: the percentage of NSA instructions executed and the percentage of CLIMB instructicns executed. Most of the NSA instructions are slightly slower than non-NSA instructions, however, the CLIMB is over ten times slower.

% CLIMB's	% Non NSA	Case Study	Coefficient "b"
.03	84.48	4	.935
.04	87.37	4	.945
.05	87.3	4	.945
.1	98.18	1	.93
.28	90.33	2	.928
.6	89.0	1	.88
.93	91.19	3	.86
1.36	90.36	2	.827

The following table provides an estimate of the 6680 performance values for the GCOS 8 environment. The coefficient "b" is an average of the first three case studies. The CP-6 measurements are not ircluded due to the unrealistically small percentage of CLIME's.

	GCOS III	GCOS 8 Accommodation	GCOS 8 Native
6680	X	.93X	.87X

3.1.3 Case Study 1

The table below summarizes the results from measurements of GCOS 8 Accommodation Mode (Ref. 1).

Instruction Type	% of Total Instructions	6680 Weight Factor	6680 Weight Value
LDP	1.2	2.0	2.40
LDD	0.18	1.9	0.342
CLIMB	0.1	20.5	2.05
EPPR	0.06	0.3	0.018
Other NSA	0.28	2.0	0.56
Non NSA	98.18	1.735	170.34
Weighted Totals			176.868

Coefficient: b = 164.4 / 176.868 = 0.930

In accommodation mode, the slave instructions are based on a single segment environment, while the master mode instructions are based on a multisegment environment. One can predict the performance of the multisegment environment by considering only the mix of master mode instructions. This is presented in the table below.

Instruction Type	% of Total Instructions	6680 Weight Factor	6680 Weight Value
LDP	7.8	2.0	15.60
LDD	1.2	1.9	2.28
CLIMB	0.6	20.5	12.3
EPPR	0.4	0.3	0.12
Other NSA	1.0	2.0	2.0
Non NSA	89.0	1.735	154.415
Weighted Totals			186.715

Coefficient: b = 164.4 / 186.715 = 0.88

3.1.4 Case Study 2

A static analysis of two Sort/Merge modules implemented in PL-6 for CP-6 is shown below. Since these modules do access multiple segments and are written in PL-6, they should provide a good indication of overall multisegment environment performance.

The first table shows the results for the Sort Command Executive, while the second table shows the results for the Tournament Driver.

Command Executive

Instruction Type	% of Total Instructions	6680 Weight Factor	6680 Weight Value
LDP	6.01	2.0	12.02
LDD	0.0	1.9	0.0
CLIMB	1.36	20.5	27.88
EPPR	1.36	0.3	0.408
Other NSA	0.91	2.0	1.82
Non NSA	90.36	1.735	156.77
Weighted Totals			198.898

Coefficient: b = 164.4 / 198.898 = 0.827

Tournament Driver

Instruction Type	% of Total Instructions	6680 Weight Factor	6680 Weight Value
LDP	7.29	2.0	14.58
LDD	0.0	1.9	0.0
CLIMB	0.28	20.5	5.74
EPPR	0.28	0.3	0.08
Other NSA	1.82	2.0	3.64
Non NSA	90.33	1.735	156.72
Weighted Totals			177.12

Coefficient: b = 164.4 / 177.12 = 0.928

3.1.5 Case Study 3

A static analysis of the Global Data Management module of ITP yielded the results shown in the table below. Though coded in GMAP, this module was chosen for the following reasons:

- it is highly structured
- it is reasonably large (5K)
- it deals with many segments so that register optimization is limited
- it is reasonably linear so that the assumption that a uniform execution takes place should be a good one

Instruction Type	% of Total Instructions	6680 Weight Factor	6680 Weight Value
LDP	1.76	2.0	3.52
LDD	1.64	1.9	3.12
CLIMB	0.93	20.5	19.06
EPPR	2.3	0.3	0.69
Other NSA	2.16	2.0	4.32
Non NSA	91.19	1.735	159.45
Weighted Totals			190.16

Coefficient: b = 164.4 / 190.16 = 0.86

This module does perhaps do more register optimization than a PL-6 generated module might. The relatively high percentage of EPPR's is due to moving the contents of one ODR to another. In a PL-6 module this would probably generate a LDP rather than an EPPR. If one-half the EPPR's are changed to LDP's, then the following figures are generated.

Instruction Type	% of Total Instructions	6680 Weight Factor	6680 Weight Value
LDP	2.91	2.0	5.82
LDD	1.64	1.9	3.12
CLIMB	0.93	20.5	19.06
EPPR	1.15	0.3	0.345
Other NSA	2.16	2.0	4.32
Non NSA	91.19	1.735	159.450
Weighted Totals			192.115

Coefficient: b = 164.4 / 192.115 = 0.86

It is interesting to note that the number of LDP's executed is of little consequence on the 6680 since the instruction time is not significantly greater than for other instructions.

3.1.6 Case Study 4

The tables below summarize the results of three CP-6 measurements. The average of the three measurements results in b=.942. See reference 4 for more information.

These measurements are not indicative of GCOS 8 timing due to the very small ratio of CLIMB's to total instructions executed, but are included for a comparison of typical instruction mixes.

Measurement_1

Instruction Type	% of Total Instructions	6680 Weight Factor	6680 Weight Value
LDP	7.25	2.0	14.5
LDD	0.0	1.9	0.0
CLIMB	0.04	20.5	0.82
EPPR	2.17	0.3	0.651
Other NSA	3.17	2.0	6.34
Non NSA	87.37	1.735	151.587
Weighted Totals			173.898

Coefficient: b = 164.4 / 173.898 = 0.945

Measurement_2

Instruction Type	% of Total Instructions	6680 Weight Factor	6680 Weight Value
LDP	7.15	2.0	14.3
LDD	0.0	1.9	0.0
CLIMB	0.05	20.5	1.025
EPPR	2.25	.3	0.675
Other NSA	3.25	2.0	6.5
Non NSA	87.3	1.735	151.466
Weighted Totals			173.966

Coefficient: b = 164.4 / 173.966 = 0.945

Measurement 3

Instruction Type	% of Total Instructions	6680 Weight Factor	6680 Weight Value
LDP	12.31	2.0	24.62
LDD	0.0	1.9	0.0
CLIMB	0.03	20.5	0.615
EPPR	1.37	0.3	0.411
Other NSA	1.81	2.0	3.62
Non NSA	84.48	1.735	146.573
Weighted Totals			175.839

Coefficient: b = 164.4 / 175.839 = 0.935

3.2 Object Code Analysis

This section compares projected multi-segment code generation of COBOL and FORTRAN in GCOS 8 with the actual code generated in GCOS-III. This discussion describes parameters of the comparison, highlights results from the comparison, and concludes with recommended future directions.

Detailed numbers are not presented.

3.2.1 Source Program Descriptions

FORTRAN — This program is a matrix inversion from a scientific benchmark. Of significant interest was the analysis of code production within the program's innermost loop. This critical code section was determined to be executed 100 million times.

COBOL-74 -- This program, obtained from a benchmark support demonstration program, heavily uses COBOL-74 string manipulation verbs -- INSPECT, STRING, UNSTRING and the PERFORM verb.

3.2.2 Types of Comparison

FORTRAN -- The program was compiled with FORTRAN-Y. It then represented the GCOS-III environment. The innerloop code was

examined and changed according to multi-segment environment requirements. This versicn then represented the GCOS 8 environment.

Those two versions were then compared in two ways. First, the GCOS-III version was compared with the GCOS 8 version assuming the hardware was constant (6680). Second, the environment was held constart (multi-segment, GCOS 8) and the hardware varied (6680 vs. ADP).

COBOL-74 -- This program was handled in the same manner as the FORTRAN program -- an existing CBL74 generated code listing was compared with a hand-coded GCOS 8 version.

3.2.3 Information Obtained

3.2.3.1 Dynamic

FORTRAN -- The innermost loop instruction count was examined in terms of number of instructions, and execution time accrual per loop trip. Execution time was adjusted for ADP pipeline breaks and cache misses.

COBOL-74 -- Since this routine contained neither iterated code or conditionally executed code, a static analysis was sufficient.

3.2.3.2 Static

For all routines the following information was recorded:

* routine name
* number lines of source
* size of produced procedure for each environment being compared
* number LDPn produced for each environment
* percent LDPn for each environment
* the average number of words of procedure code generated for each procedure statement for each environment.

3.2.4 Results

* The COBOL-74 sample had a relatively low level of LDP's (1%). A COBOL-74 sample with more parameter passing would generate more LDP instructions.

* The COBOL-74 sample has a potential problem regarding excessive ADP pipeline breakage during execution of subroutine and library call linkage.

* The COBOL-74 multi-segment sample also has an excessive ADP pipeline breakage problem for PERFORM code generation. This problem results when exiting a perform block, i.e., "TRA to a TRA".

* COBOL-74 generated code for the multi-segment environment (ADP) should execute above the ADP 6X baseline.

* Degradation from the GCOS-III performance baseline should be less than 10% for COBOL-74 generated code. Since COBOL-74 has no "dynamic" pointers except passed parameters, it is essential to continue global register assignments for parameter addresses and further to extend COBOL-74 to subject base pointers (to working storage, process area, etc.) to the same register management as parameter addresses.

* The FORTRAN sample also had a relatively low number of LDPs (2.3%). Again, this was partially due to the nature of the language (no cynamic data segments), however much credit to reducing this figure must go the the FORTRAN-Y optimizer, as there was parameter passing of signicance (4 per subroutine call). The LDPs for the parameter address loading totaled 400 (4 parameters * 100 calls to the matrix inversior routine). However, references to these parameter address values totaled within the innerloop one million. Thus, the ratio of loads to use was quite low, as well as being well separated.

* FORTRAN generated code for the multi-segment environment (ADP) should have no problem meeting the ADP 6X baseline performance goal. In fact, after accounting for cache misses and pipeline breaks, this inner loop code improved in excess of a 10X factor.

* There should be minimal degradation when comparing GCOS-III environment code productions with GCOS 8 multi-segment environment code productions. The innerloop code in particular should not degrade at all since there are no calls and no LDPs are within it. The degradation, if any, would result from the new calling sequence and the global pointer register loads upon each entry to the matrix inversion. However, as previously stated, those events only occur 100 times.

3.2.5 Conclusions

1. From the small number of programs examined it would appear that both FORTRAN-Y and COBOL will meet the performance goals for the code generated by the compilers. More work needs to be done to quantify performance in this area as well as in the linkage to the I/O support routines, which was not analyzed. Suggestions for this are included in the recommendations.

2. Global optimizing compilers will have a significant effect on performance. This is true on the 6680 where an optimizer would reduce the number of Load Pointer instructions generated and executed. It will be even more significant on the ADP where an optimizer would take advantage of the pipeline as well as reduce the number of Load Pointer instructions.

3.2.6 Recommendations

1. Plans should be put in place to add optimizers to all language translators which do not have them.

2. Possibilities of a binder changing/adding/deleting instructions in addition to relocating addresses should be studied. The possibilities include adjusting of address fields in conjunction with the removal of address register manipulation and recognizing references to bound segments and changing references as a result. The extreme to which this can be utilized is to bind programs and data into a single segment.

3. Improve the code generator in the COBOL-74 compiler to improve the instructions generated for both PERFORM and CALL.

4. Perform studies on GCOS-III by inserting pulse instructions into the call and entry operators to determine as much of the information on performance factors as possible and compare those with the hand calculated numbers for the recommended model.

4.0 Migration

*** To be supplied ***

SECTION 5

DETAILED SPECIFICATIONS

1.0 Control Structures

To be supplied.

2.0 Interface Conventions

To be supplied.

APPENDIX A

ADDRESS REPRESENTATION

1.0 Introduction

The representation of address values is the central problem in the design of the micro-structure for the multi-segmented run-time environment. The use of address values is widespread and interacts strongly with the overall system goals.

Address values are required in the implementation and control of:

- Exception Processing
- Memory Management and Software Stacks
- Arguments and Parameter Referencing
- Locate mode input-output
- List structure processing
- Connection to run-time support

In high order languages, these facilities show up as distinct language constructs for:

- Pointers and Based Storage
- Entries and Labels
- Alternate returns ard Exception conditions

Both the facilities and their high order language constructs appear in Honeywell and customer software. Their wide usage is reflected in the large number of system goals that are related to the choice of address value:

- Job performance and throughput
- Uniform micro-structure
- User visible address values
- Automatic space allocation and recursion
- Exception processing
- Large procedures and data spaces
- Support of virtual environments
- Support of shared elements
- Provide program and data integrity

Use current hardware
Minimize ITP conversion

In the design process, these several aspects of address values were reduced into the following set of design criteria:

* The address value must be storable in user data space. The address value representation must be a legal slave space data format.

* The address value representation must allow uniform reference to domain-external parameters and domain-local data.

* The substantive address value must retain its identity throughout being loaded into and stored from an Operand Descriptor Register (ODR).

* The address value representation must support bit level addressability for operands.

* It is desirable that the address value representation use hardware with relatively high performance.

* It is desirable that the address value representation support segment level content integrity.

* It is desirable that the address value representation support an addressability greater than 256K words.

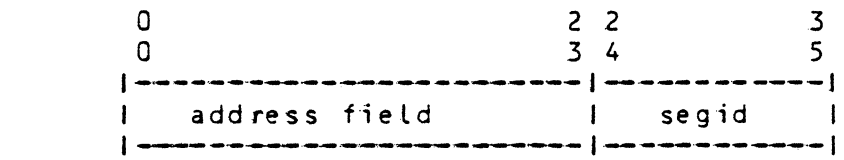
* It is desirable that the address value representation support domain structures containing more than 1024 segments.

* It is desirable that the address value be valid across domains.

The following sections discuss the alternate address value representations investigated.

2.0 NSA Pointer

The NSA pointer is a hardware single word data format containing an address field and a segment identifier. The word has the following format:



The address field, bits 0-23, has the same format as an address register and gives a word, byte, and bit offset into the associated segment. The segid, bits 24-35, identifies the descriptor segment and entry value for the descriptor framing the associated segment. The segid may reference only the linkage descriptor segment, the argument descriptor segment, or the parameter descriptor segment. The size of the segid field allows a maximum of 1024 entries in each of the three descriptor segments.

There are several drawbacks to the use of the NSA pointer as the address value representation:

1. The hardware instruction for loading a pointer (LDP) is not one of the faster NSA instructions. The instruction is inherently slow, since it includes a second memory access to acquire the NSA descriptor referenced by the segid value.

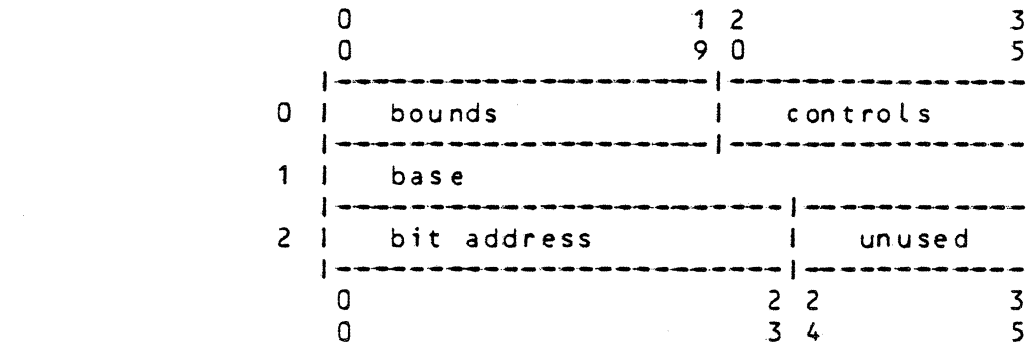
2. The address field of 24 bits limits the useful offset value to a segment of 256K words.

3. The NSA pointer is effectively limited to a domain of 1024 segments. A domain is defined by the contents of the linkage descriptor segment. The segid value makes direct reference to the linkage segment and is limited to 1024 entries.

4. The NSA pointer value is not valid across domains.

3.0 NSA Descriptor and Address Register Value

The NSA descriptor is a hardware double-word data format identifying the location and extent of a segment. Since both location and extent are in terms of bytes, the NSA descriptor must be accompanied by an address register value to give the bit address of the operand. This three word address value representation would have the format:

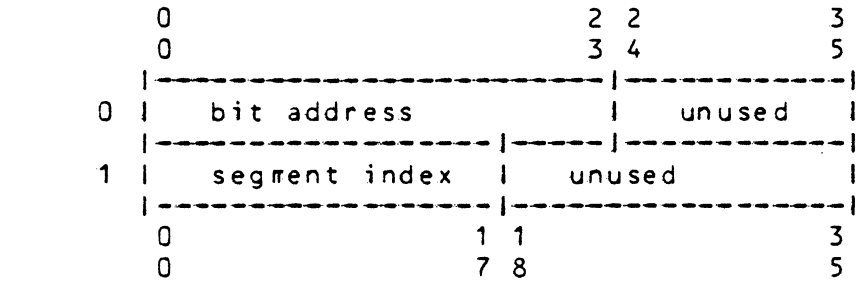


The bounds field, bits 0-19 of word 0, contains the maximum valid byte address within the segment. The control field, bits 20-35 of word 0, identifies the working space within which the segment resides and contains access control information. The base field, word 1, locates the byte offset of the segment within the working space identified by the control field. The bit address, bits 0-23 of word 2, has the same format as an address register and gives the word, byte and bit offset of the operand within the segment.

Using a NSA descriptor plus an address register value as an address value representation has the disadvantage that such a construct cannot usefully be stored into user data space. Although the address register value and descriptor content can be stored into operand space, the descriptor cannot be loaded into a descriptor register from operand space. Separation of the address register value in operand space from the descriptor in a special descriptor segment raises insurmountable problems in synchronizing the two spaces and passing address values between procedures witnin a domain.

4.0 Segment Table Index

When represented as a bit address and an offset (index) into some descriptor segment, the address value representation would require a two word construct:



The bit address field, bits 0-23 of word 0, has the same formst as an address register and gives the word, byte, and bit offset within the associated segment. The segment index, bits 0-17 of word 1, contains a value that identifies the proper descriptor within an associated descriptor segment. The descriptor segment would have to be located by a descriptor at some cancrical position in the linkage segment. To maintain efficiency, at least one ODR would have to be dedicated to framing this descriptor segment.

There are at least three drawbacks to the use of this format for the address value representation:

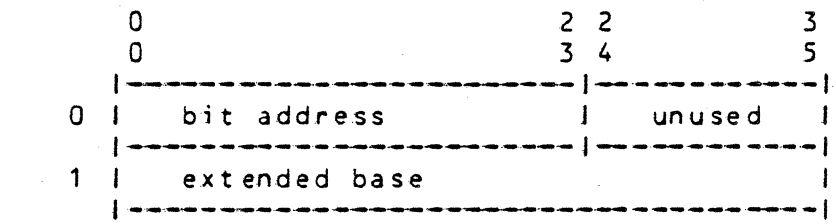
1. The value of the address value cannot be maintained across the loading cf an ODR. The associated descriptor can be placed into an ODR and the bit address value can be placed in the matching address register. There is no place, however, in which to remember the segment index value. There is no way, therefore, in which the address value can be reconstituted in data space.

2. This format is not especially efficient, recuiring several instructions tc load the ODR and address register.

3. This representation is not valid across domains.

5.0 Super Pointer

The super pointer representation of an address value is predicated on all of the data segments in a domain, regardless of access controls, being collected into a single super segment. The super pointer locates data within this super segment. The super pointer is envisioned as a two word construct:



The bit address field, bits 0-23 of word 0, has the same format as an address register and gives the word, byte, and bit offset from the base value within the associated super segment. The extended base field, word 1, gives the byte offset of a datum within the associated super segment. The super segment itself would have to be located by a super descriptor at some canonical position within the linkage segment. To maintain efficiency, at least one ODR would have to be dedicated to framing the super segment.

There are several drawbacks in using a super pointer as the address value representation:

1. Address values represented as super pointers stored in user data space can point only into the super segment. Parameters passed into a domain are not in the super segment. Therefore, parameter address values cannot be represented via super pointers.

   Moving parameter values into the invoked domain and back to the invoking domain is inefficient.

   Creating self-describing super pointers is inefficient in that each pointer would have to be tested for type before being utilized.

2. The collection of all data segments of a domain into a single super segmert vitiates any attempt to control the access to particular segments or classes of data. All of the content of the super segment would have the same access permissions as the most public datum in the super segment.

3. The super pointer is not valid across domains.

6.0 Evaluation Summary

The following table summarizes the attributes of each possible address value representation against the stated design criteria.

CRITERION	RANK	NSA PTR.	DESC. ADDR.	TABLE INDEX	SUPER PTR.
Storable in data space	must	Y	N	Y	Y
Uniform parameter referencing	must	Y	Y	Y	N
Retain identity through CDR	must	Y	Y	N	Y
Bit level addressability	must	Y	Y	Y	Y
Relatively high performarce	1	N	Y	N	Y
Support segment integrity	1	Y	Y	Y	N
Addressability greater than 256K	2	N	Y	Y	Y
Domains exceeding 1024 segments	2	N	Y	Y	Y
Valid across domains	3	N	Y	N	N

Only one alternative, the NSA pointer, supports all of the absolute requirements.

APPENDIX B

DESIGN SOURCES

The following sections describe alternative program environment definitions considered as input to the GCOS 8 design process.

1.0 Multics Program Environment

1.1 Objectives of Multics Runtime Environment

- Ease of program development
  - Considered major and increasing factor in computer expense
  - System developers as well as end users
  - Large address space
  - Minimum pre-specification
  - High level language support
- Efficient execution
  - Minimize copying
  - Minimize main storage requirement
- Protection
  - User from himself
  - User from other users
  - System from user
  - Accident or malice
- Resource administration and control
  - Centralizable or delegatable
  - Automatic
  - Flexible
- Adaptability
  - To different needs of different users
  - To varying scale configurations
  - To future requirements
  - To future technology
    - New devices
    - Declining cost of storage
    - New programming languages and techniques

1.2 Features of Multics Runtime Environment

- Process is fundamental structure
- Addressing mechanism
  - Memory size limits
    - 256K words per segment
    - 4094 segments per process
  - Segmentation
    - Hardware supports use of segment number and offset
      - Pointer registers
      - ITS indirect word pair
      - Implicit use of Procedure Segment Register
    - Uses of segments
      - Procedure segments
        - Single compilation, directly executable
        - Bound segment, same format as compiler output
        - Pure procedure
      - Data segments
        - Process private
        - Shared
      - Supervisor procecure and data
  - Paging

- Invisible to user
  - All instructions and modifiers work across page fault
- Segments don't share pages
- Page mapped 1:1 with disk record; memory encaches disk
- Segments can grow
  - Zero pages in segment allocated when stored into
- Uses of paging
  - Efficient buffer allocation
  - Configuration incependence of user programs

Security and integrity
- Access control per-segment, per-user
  - Derived from information in file system
  - Updated immediately if information changes
- Access control dimensions
  - Intraprocess access control: rings
    - Rings 0-7, 0 most privileged (central)
    - Brackets for write, read, execute, call
    - Hardware validation of ring number in pointer
    - Uses
      - Protection of supervisor from user
      - Running a program in an isolated environment
      - Providing controlled use of data or program
  - Per-user access control: Access Control List
    - Modes Read, Execute, Write
    - Enforced by hardware on every reference
    - Uses
      - Read sharing: use of common data and program
        - Memory and channel efficiency
        - Coordination of user activity (library)
      - Write sharing
        - Process synchronization
  - Nondiscretionary access control: Access Isolation Mechanism
    - Level and category, like military security
    - Uses
      - Prevention of accidental disclosure of information
  - Defaults make access control transparent for common case

Generated code
- Stack segment (per-ring) for program temporary storage
  - Stack header has environment definition pointers
  - Hardware knows stack segment number and register convention
    - Does not know any fixed offsets in stack
  - Recursive code stardard
- Threaded code with operator segment references
  - Operator segment shared by multiple processes, all rings
  - Operator segment located by language convention at entry
- Standard object segment format
  - Text (instructions and constants)
  - Definitions (inward reference)
    - Entrypoint
    - Argument descriptors for each entrypoint
  - Symbol (optional)

- Static template (optional)
- Break map (for debuggers) (optional)
- Object map

Linkage section per ring
- Contains ITS pointers or fault pairs
- Supplies segment number for external reference

Static storage area per ring
- Allocated in the same place as linkage

Standard call operator
- Used even by assembly language programs, via macro

Standard argument list
- Header
- Argument pointers
- Argument descriptors

Standard data representation
- ASCII character set
- Machine-supported data types
- Array and string representation
- Packing and alignment
- Pointers
  - Implemented as ITS pair, can use for indirection
  - Packed pointers
  - Ring number in pointer in storage

Supervisor

Name management

Segments searched for by symbolic name
- Assigned segment number and made known
- Subsequent searches for same object very efficient
- Per-ring search rules control search for object
- Referencing directory rule helps subsystem packaging
- System command internals available to user
- Site may modify default search rules

Linkage and name space not reset implicitly
- No job step or command concept
- Run units, explicit termination optional

Interprogram linkage
- Dynamic linking standard
- Binding optional
- Prelinking optional
- Unlinking of dynamic link on demand
- Run units

Supervisor call and return
- Same mechanism as any other call
- Inner ring programs take some care not to be subverted

Exception handling

Error indication
- Symbolic error codes only, numeric values sealed
- Convention is to use final argument of subroutine
- Standard I/O stream for error messages

Query handling

Signal mechanism
- Condition handlers

```
      Cleanup handlers
      Any_other handlers
      Cross-ring signalling
      Static condition handlers
      Hardware faults handled as signals
      QUIT handled as signal
      Default environment action
        New command level
        Start, release
    Process termination
      Epilogue handlers
Replace parts of environment
    Subset
    Extend
    Test new version
Uniform execution regardless of input stream
    Stream I/O system
    Resource Control Package
      Symbolic resource names
    User may create outer module
    User may generate DCWs for I/O
```

2.0 CP-6 Program Environment

2.1 Goals

The goals of CP-6 are tc provide an attractive upgrade alternative to Sigma series hardware users by offering a system personality very similar to CP-V. Conversion of data files in format and from EBCDIC to ASCII was assumed necessary but all file system and file access method functionality of CP-V was copied as closely as pcssible.

Development of the system was to be based on using a higher order language for most cf the implementation. Both external (command language) and internal (calling sequence) uniformity were given early attertion and high priority.

Compatibility with GCOS was explicitly of secondary importance.

2.2 General Characteristics

CP-6 uses the same NSA hardware as GCOS 8 but in a much more limited way. Although stcrage managemment takes advantage of the page tables, dynamic paging is not supported. The domain structure of user programs is fixed and the user address space is limited tc a total of 398K words.

Approximately 90% of system software is written in PL-6. Uniform interface conventions are strongly enforced.

The system is seperated into four domains and each user has a single domain. The system domains are:

Monitor
Command Processor
Interactive Debugger
Alternate Shared Library

2.3 Process Structure

Each user process is assigned a Working Space of fixed structure.

- o A fixed page table space allocation limits each WS to 512K words.
- o The user program can access at most 398K of the WS.
- o There is one domain having a fixed segment structure.
- o Segments addressatle by the user process are:
  - Instruction Segment; 256K maximum size.

- Control Segment; 14K maximum and Read-Only.
- Data Segments; at most 8 segments totalling 128K.

o The WS page table is used for memory control.
  - Certain portions are pre-assigned for functions.
  - Process and/or constant data sharing is achieved by mapping the same physical page into two or more WS page tables. (32K of user instruction segment is normally reserved for a "library" of shared pure procedure.)
  - The user program may request dynamic allocation of pages in the page table.
  - Overlay usage in the Instruction Segment is supported.

2.4 Program Structure

Compiler output is not directly executable but must be linked with its supporting subroutines into a run unit.

Large programs must be overlay structured in a conscious way.

Intra-domain procedure calls are implemented using TSX instructions.

The user domain may CLIMB to the Alternate Shared Library and reaches the Monitor by PMME.

The user process may directly manipulate pages:

- allocate and free data segment space in words,
- allocate and free real Instruction Segment pages other than those allocated via the linking process,
- Allocate and free virtual Instruction Segment pages other than those allocated via the linking process.

Subroutines may be shared only by being page mapped into the top 32K of the Instruction Segment.

User procedures may be shared via special post-linking processing to identify them as sharable elements.

Hardware pointers and vectors are usable via both assembly language and PL-6.

Compiler output segregates data and procedure. Pure procedure is created by PL-6 to allow sharing.

2.5 Exception Processing

The standard calling sequence provides for an alternate return point.

A one-way inter-procedure exception path is supported by the PL-6 REMEMBER/UNWIND feature.

Process level exceptions (ON conditions) are supported via ASYNCHRONOUS procedures and the M$XCON (exit control) facility of the operating system.

2.6 System Personality

A single JCL provides for both batch and interactive usage modes.

The same system interface mechanism is available to programs in batch and interactive execution modes.

The same I/O mechanism works for both batch and interactive programs. The inteactive state may be determined from file attributes.

System search rules are the same for JCL and programmatic procedure invocation.

System modules are dynamically replaceable without system interruption.

3.0 GCOS 8 SR 1000 Program Environment

3.1 Brief History

The native environment present in SR 1000 is partially inherited from ACOS V1.1 which was based on a Toshiba-HIS joint design effort goirg back to 1975 and 1976. It goes beyond that base in many ways including checkout of the multiple Shared Run Unit Library capability and the addition of a limited capability Dynamic Linker.

3.2 Goals

The goals of the release which apply to the native environment include:

- o Overcome limitations of GCOS-III
  - Slave memory size
  - Files per activity (increased PAT space)
  - Program number limit
  - SSA module fragmentation
  - Memory fragmentation (compaction overhead)
- o Utilize NSA hardware features
  - Optimize real memory utilization
  - Improve integrity and security
- o Support the Integrated Transaction Processing system

3.3 General Characteristics

The unique address space of a program is in a private Working Space "viewed" through WSR 7. This process-local Working Space is divided into control information storage (the process structure) and program storage. The first 64K virtual addresses are reserved for control information and descriptor storage. All used pages in this area must be memory resident when the process is not swapped out.

Dynamic paging of both program and shared address spaces is supported by ruling out unsupportable instruction sequences. Explicit overlay management is not supported by the system in native mode.

Program construction facilities treat each compile unit as a domain. Thus all runtime services are invoked by a CLIMB instruction.

Segmentation is assumed in program construction and is always based on standard descriptors. This means that the largest segment size is 256K words.

Operating system services are reached either by a CLIMB or a PMME instruction. The interfaces are not consistent in style and some have undesirable features such as passing codes in registers.

3.4 Sharing Mechanisms

Addressability to Sharec Working Spaces is available to a native made program via WSRs 2-6. Content of these Shared Working Spaces may be loaded by an unrealesed utility to Working Spaces having fixed relationshios to the WSRs. These relationships, the status of the contents, and other information is recorded ina hard-core table.

An unreleased utility provides optional static linking to Shared domains but Run Units so linked are vulnerable to changes in the content of Shared Working Spaces to which they are linked. Compatibility of a Run Unit with the Shared Working Spaces available at the time of its execution is checked tc prevent a mis-match.

Alternatively, dynamically assigned Working Spaces may be loaded by a loader program which is part of the developmental scaffolding used by ITP. The Shared Working Spaces in this method are controlled by a "sleeping" process which holds the Working Space, backing store, etc.

A primitive Dynamic Linker supports linking to ITP shared software.

3.5 Process Structure

User program (process local) virtual address space may be at least 1.6 million words.

More than 25C files may be assigned to an activity.

Construction of both user and shared programs is completely flexible (within hardware constraints) in the use of multiple segments and multiple domains.

4.0 GCOS-IV, June 1979 Program Environment

4.1 Brief History

This environment definition was developed primarily within the Language and Database organization in 1978 and 1979 to provide a base for compiler planning and particularly to establish a target environment for the development of PL-6 for GCOS 8. It is a slight extension of the SR 1000 environment in that new approaches are taken to the construction of programs which free the system designer to construct domains from multiple compile units.

Certain conventions worked out during development of this specification became part of SR 1000. In particular, domain structure, null descriptor, null pointer and revised exception processing conventions were adopted.

4.2 Goals

Definition of the execution environment as seen by a compiler code generator was the fundamental goal of this effort. Support of all general features of higher order language systems, efficient inter-module calling sequences, and maximum uniformity of conventions were considered of highest priority.

Maximum generality of sharing, uniformity, and ease of use were also taken as important goals.

4.3 General Characteristics

The most significant variation from the program construction available previously for native mode is the assumption that modules generated by a compiler would normally be combined with others in a single comain. This choice was made in order to employ the hardware "pointer" datum as the "pointer" data type of several language systems. It also led to a means of providing intimate run time supporting software that could be shared without the use of the CLIMB instruction.

A generalization of dynamic linking was envisioned in which symbolic information in every domain would provide names to be matched against directories in each Shared Working Space.

Dynamic association of Shared Run Unit Libraries with Working Spaces and of process with Shared Working Spaces was proposed but not fully defined in the specification.

5.0 An Environment Modeled on Multics

This note describes briefly how to adapt the Multics multi-segment runtime environment to the NSA machine, in order to create a GCOS-IV multi-segment runtime environment.

5.1 Fundamental Mapping

A Multics segment will be mapped into an NSA segment.

Pointer values will be represented by NSA pointers.

Multics rings will be approximated by NSA domains.

Each domain will have a descriptor segment; segment numbers in all domains of a process will refer to the same segment, with possibly different access rights.

5.2 Detailed Description

5.2.1 Segment Number Assignment and Pointers

Pointers can be shared between domains only if the segment numbers are assigned identically in both domains. The Multics approach to this problem involves several rules:

1. Pointers are never valid after shutdown and reboot.

2. Pointers are valid across processes only in a special case: system-wide assignments of segment numbers to supervisor segments at bootload time. Thus, a pointer to a supervisor segment is valid in all processes.

3. Pointers are freely passed within a process, but it is the process's own resposibility to garbage collect pointers within a domain (ring). That is, a process can construct a pointer, hide it somewhere, and release the segment number; the pointer is invalid but the system does not automatically invalidate the pointer.

Segment numbers 0-N will be reserved for supervisor segments in all processes (N set at bootload time). Then, segment numbers N+1 to N+M will be reserved for per-work station segments, where M is variable according to work station and determined at process creation time. The rest of the segments in the process are assigned segment numbers on a first-come, first-served basis.

This does not preclude two processes sharing a procedure segment, assigning it different segment numbers in different processes. The procedure will, however, require a linkage

section which is impure and per-domain which contains segment numbers needed by the executable code to refer to its environment and for inward reference.

5.2.2 Structures Adopted from the Multics Environment

Stack
Stack frame
Linkage Offset Table
Combined Linakge Area
Reference Name Table
Known Segment Table
Argument list header
Argument list
Argument descriptor

5.2.3 Procedure call

One procedure will call another according to the following scenario. (Suppose A calls B):

1. Procedure A prepares an argument list for B in A's stack frame.

2. Prccedure A obtains a pointer to procedure B.

3. Procedure A enters the CALL operator.

4. The CALL operator saves the return point in A's stack frame and enters prccedure B.

5. Prccedure B performs a standard entry sequence which
   - Builds a stack frame for B
   - Establishes addressability for B's linkage section
   - Establishes addressability for B's arguments

Arguments will be passec as they are in Multics, not via CLIMB. The argument list will be a list of NSA pointers to argument values, stored cn the software stack. If CLIMB is used, it will not be usec for argument passing. The parameter stack is not used. The only CLIMB opcode will be in the operator segment which contains the call operator.

5.2.4 Compiler Output

Pointers may be passed between domains. The output of a compiler is a file which contains several sections: executable ccde, linkage definitions, linkage section template, symbol section, ard object map. Output from separate compilations can be combined into one segment by a "binder."

5.2.5 Differences from Multics due to NSA

Compiler output is pure procedure, threaded code. A process may have many domains, but there is a limit of 1024 segments per process. (Multics had this limit for many years: most processes still operate within it. Administrative action can increase the size to 4096 for special processes.)

Pointers do not carry a ring number. This requires that all pointers input to a domain be validated by the callee. Such code was once written for 645 Multics: its construction is fraught with subtleties and dangers. On the other hand, we understand the problem.

Since NSA does not provide an ITS pointer, all indirect addressing must be replaced by explicit register loading.

The size of a segment is 256K words, same as in Multics, unless super descriptors are used. These can be used if there are some limitations, like only one per process.

5.3 What Must be Built

5.3.1 Supervisor Services

Name Management

Per-domain reference name management.

Per-process and per-work-station segment number management.

Per-ring search rules.

Interprogram Linkage

Dynamic linking
Unlinking of dynamic lirk on demand
Run units

Supervisor call and return

Write-arounds to GCOS-8 functions must be provided so that the user program can call upon the supervisor by a language call instead of via a MME. The supervisor routines must take some care not to be subverted if pointer arguments are passed.

Exception Handling

Software convention must be established for
error code

Standard I/O stream fcr error messages
Query handling
Signal mechanism
  Conditicn handlers
  Cleanup handlers
  Any_other handlers
  Cross-ring signalling
  Static condition handlers
  Hardware faults hancled as signals
  Default environment action
QUIT
Process termination
Epilogue handlers

5.3.2 Language_Support

Standard_Operators

The standard call, push, and return operators must be written. If multiple operator segments are permitted in a domain then the conventicns for making the various operator segments addressable must be worked out.

Linker

The standard linker must be designed. This requires the following pieces:
Fault handling
Definition search
Linkage space assignment
Linkage template loading
Process restart

Binder

A binder will be requirec for the initial release, in order to conserve segment numbers.

Interface_to_language_runtime_I/O

Efficiency of the COBOL and FORTRAN I/O packages will be important, and special care must be given to making this function efficiert.

6.0 Other Inputs

TBS by GA Mann.

APPENDIX C

DESIGN EVALUATION

The GCOS Multi-Segment Runtime Environment Committee evaluated the alternative design strategies proposed for the GCOS 8 MSRTE and chose an approach which was an evolutionary development from the current 4VX product. The major reason for this choice was the feeling that no other approach could be implemented for delivery at the end of 1981.

1.0 Major Approaches

1.1 Multics Approach

The Multics approach is a low risk approach to satisfying most of the functional design objectives for the runtime environment; we know this because Multics satisfies most of these objectives and already works. Performance parity with GCOS-III is probably not possible with this approach, or any other approach considered: but predicting the performance of a Multics-approach environment was not pursued in detail.

The amount of code to be written for the Multics approach is known to be large; compiler code generators for all compilers, binders, linkers, and supervisor services must be built as described above. This amount of code is about the same for all proposed implementations, but the additional work for the Multics approach would be the re-implementation of 4VX/ITP and other GCOS code to work with the new environment.

This approach was not given a large amount of consideration. Once we determined that the complete job was very large, and could not be reasonably promised for end of 1981, we turned to other schemes. Another reason we did not pursue this approach too far was that it used NSA pointers and the LDP opcode heavily, and at the time we were in hopes of discovering an approach which did not suffer from the performance problems of this method.

1.2 CP-6 Approach

We really didn't evaluate this approach, because CP-6 does not provide sufficient support for large address spaces. The question of how to alter the CP-6 environment to support larger address spaces was not investigated.

1.3 GCOS 8 SR 1000 Environment

Although code generators for all languages, an A-unit merger, and a B-unit binder must be written, it is possible that some use may be made of the existing loader, and the dynamic linking and memory management software used by ITP.

Compared to the Multics approach, this method might be less code, or it might be more, depending on how much old code can be adapted to the new circumstances. It is definitely more design: many complex features of the RTE would have to be invented, which could be copied from Multics if we took the Multics approach.

If we assume that ITP is going to be kept with minimum change, then the desire for a uniform environment will have a strong influence on the shape of the MSRTE. Several strategies used by ITP, such as process structure, memory management, per-opening domains for every use of a file, cannot be accommodated within many of the possible RTEs.

Some of the committee members expressed the strong desire to avoid any canonicalization of domain internals; that is, it would be possible to have many different internal structures in different domains. This was advanced as an advantage to program developers since the effects of an error would not propagate.

1.4 GCOS-IV June 1979 Environment

This environment was considered to be a minor variant of SR 1000 and our eventual design adopted features as appropriate.

APPENDIX D

COMPETITIVE COMPARISON

TBS

MULTI-SEGMENT SHARED RUN-TIME ENVIRONMENT

- GOALS & CONSTRAINTS
- MACRO-STRUCTURE
- MICRO-STRUCTURE
- PERFORMANCE

4/18/80

G O A L S

GOAL	RANK	RESPONSE
MIGRATE WITHOUT HOL SOURCE CHANGES	MUST	YES
ACCOMMODATE GCOS-III EXECUTABLE FORMATS	MUST	YES
ACCOMMODATE GCOS-III TSS, TDS, & DMIV-TP	MUST	YES
JOB PERFORMANCE AT LEAST 90% OF GCOS-III	MUST	TBD
THROUGHPUT AT LEAST 90% OF GCOS-III	MUST	TBD
USER VISIBLE ADDRESS VALUES	MUST	YES
AUTOMATIC SPACE ALLOCATION AND RECURSION	MUST	YES
EXCEPTION PROCESSING	MUST	TBD
DYNAMIC SUBPROGRAM INVOCATION	MUST	TBD
SUPPORT LARGE PROCEDURES	MUST	YES
SUPPORT LARGE DATA SPACES	MUST	TBD
SUPPORT DISTRIBUTED SYSTEM ARCHITECTURE	MUST	YES
SUPPORT A VIRTUAL ENVIRONMENT	MUST	YES
SUPPORT SHARED ELEMENTS	MUST	YES
PROVIDE PROGRAM AND DATA INTEGRITY	MUST	YES
PROVIDE USER ACCESS CONTROL	MUST	YES
USE CURRENT HARDWARE	MUST	YES

G O A L S (CONTINUED)

GOAL	RANK	RESPONSE
UNIFORM MICRO-STRUCTURE ENVIRONMENT	1	YES
UNIFORM MACRO-STRUCTURE PERSONALITY	2	YES
PROCESS SYNCHRONIZATION	2	YES
SUPPORT LARGE NUMBER OF FILES	2	YES
SUPPORT LARGE NUMBER OF TERMINALS	2	YES
SUPPORT MULTIPLE VERSIONS OF SAME MODULE	2	YES
SUPPORT DYNAMIC SOFTWARE INSTALLATION	2	YES
EXTENDIBLE TO FUTURE PRODUCT DIRECTIONS	2	YES
SUPPORT ARRAYS LARGER THAN 256K	3	NO
PROTECT HONEYWELL PRICED SOFTWARE	3	TBD
MIGRATE WITHOUT ASSEMBLY SOURCE CHANGES	4	TBD
TASKING	4	TBD

CONSTRAINTS

RELEASE WITH 5V

MINIMIZE CONVERSION

USE CURRENT HARDWARE

SUPPORT HIGH ORDER LANGUAGE FUNCTIONS

SUBCOMMITTEES

TWO SUBCOMMITTEES WERE FORMED.

1. MICRO-STRUCTURE SUBCOMMITTEE

   DEFINE THE INTERNAL ENVIRONMENT

   - o THE STRUCTURE INTERNAL TO A DOMAIN
   - o CALLING SEQUENCES WITHIN & BETWEEN DOMAINS
   - o LINKAGE SEGMENT LAYOUT

   MEMBERS:

   DICK WILSON (CHAIRMAN), JOHN WERTZ, TOM VAN VLECK, FRANK LITTLE

2. MACRO-STRUCTURE SUBCOMMITTEE

   DEFINE THE EXTERNAL ENVIRONMENT

   - o EVERYTHING EXTERNAL TO THE DOMAIN - PROCESS STRUCTURE, OBJECT UNIT AND RUN UNIT STRUCTURE, WSQ STRUCTURE, . . .
   - o RUN TIME SUPPORT SERVICES - DYNAMIC LINKER, ETC.
   - o PROCESS & DOMAIN CREATION MECHANISMS
   - o SHARING MECHANISMS JCL, PROCESSORS, . . .
   - o HANDLING OF WSR'S & SEARCH RULES

   MEMBERS:

   CHARLIE COFLIN (CHAIRMAN) GEORGE MANN, AL BEARD

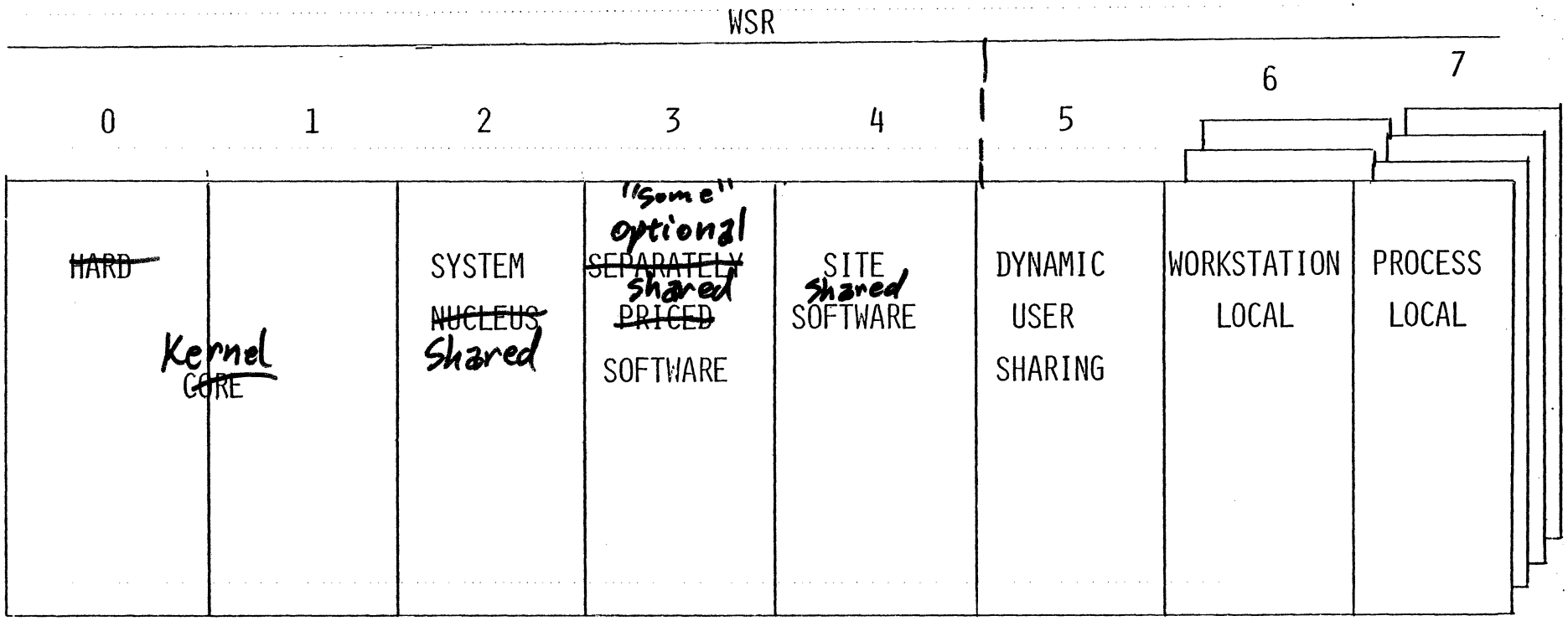
EXTERNAL ENVIRONMENT

1. USAGE OF WORKING SPACE REGISTERS TO SUPPORT SHARING

2. TYPES OF SHARING

3. CONSTRUCTION OF THE VIRTUAL ENVIRONMENT

WSR USAGE FOR SHARING

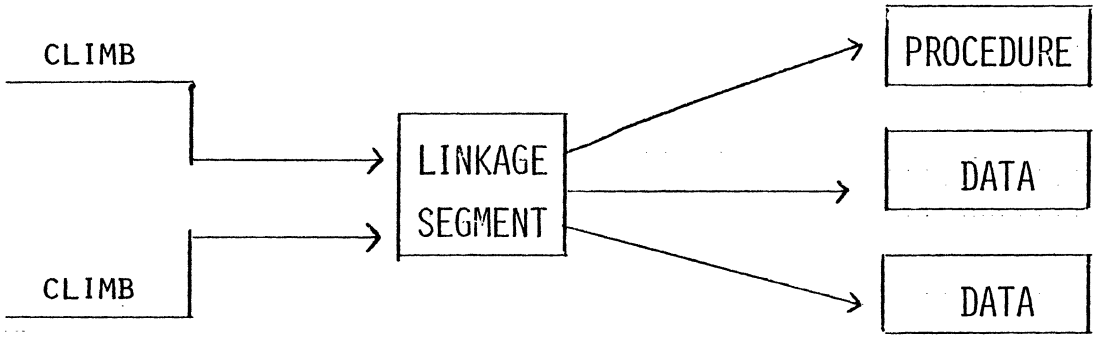


WSR~~Ø~~ - WSR~~7~~4 ARE COMMON TO ALL PROCESSES

4/18/80

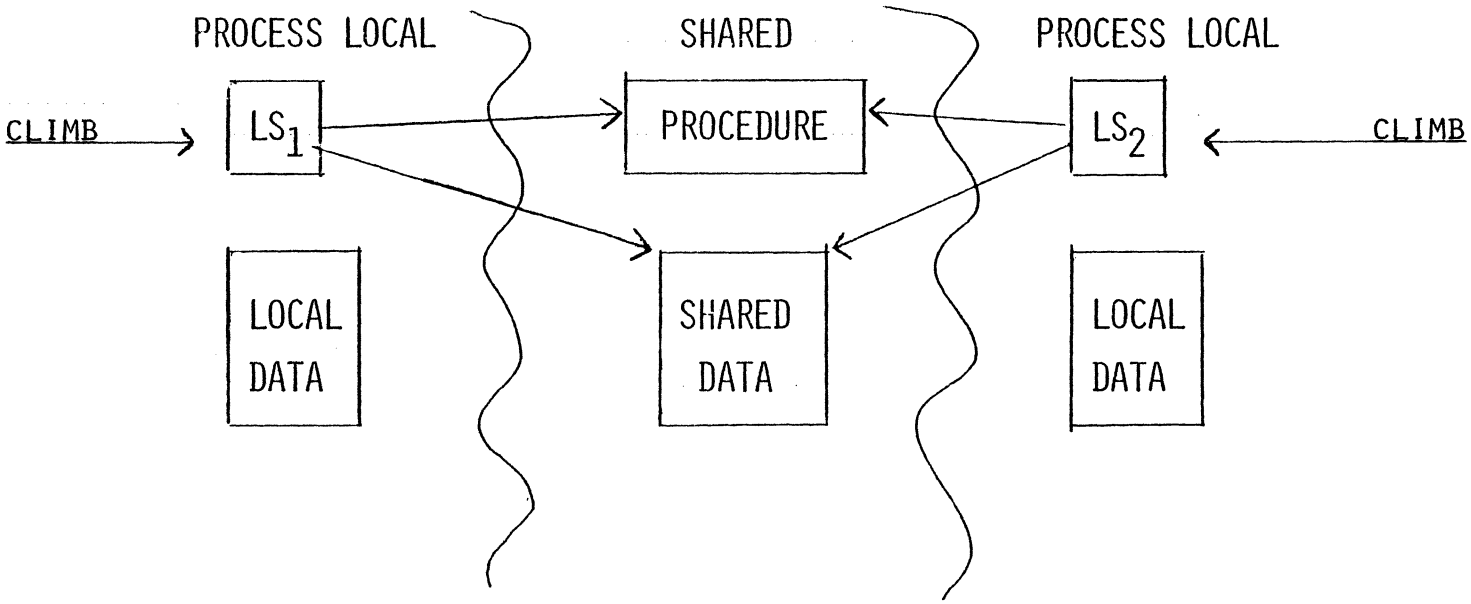
TYPES OF SHARING

1. DOMAIN INSTANCE SHARING
   - LINKAGE IS SHARED
   - ALL SEGMENTS OF DOMAIN ARE SHARED



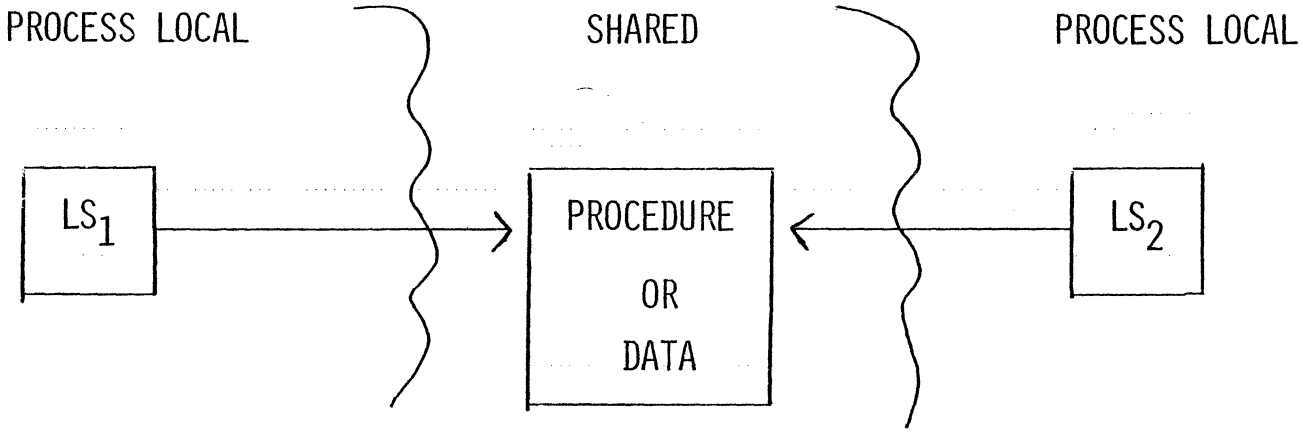
TYPES OF SHARING (CONTINUED)

2. DOMAIN PATTERN SHARING
   - LINKAGE SEGMENT IS NOT SHARED
   - DESCRIBES BOTH SHARED AND UNSHARED SEGMENTS



3. SEGMENT SHARING

- IF PROCEDURE SEGMENT, THEN:
  - o ALL DATA REFERENCES ARE TO PARAMETERS OR DYNAMIC DATA
  - o PURE PROCEDURE
- IF DATA SEGMENT, THEN:
  - o MUST BE GATED



descriptors may not be in the same offset

DEF - A-UNIT

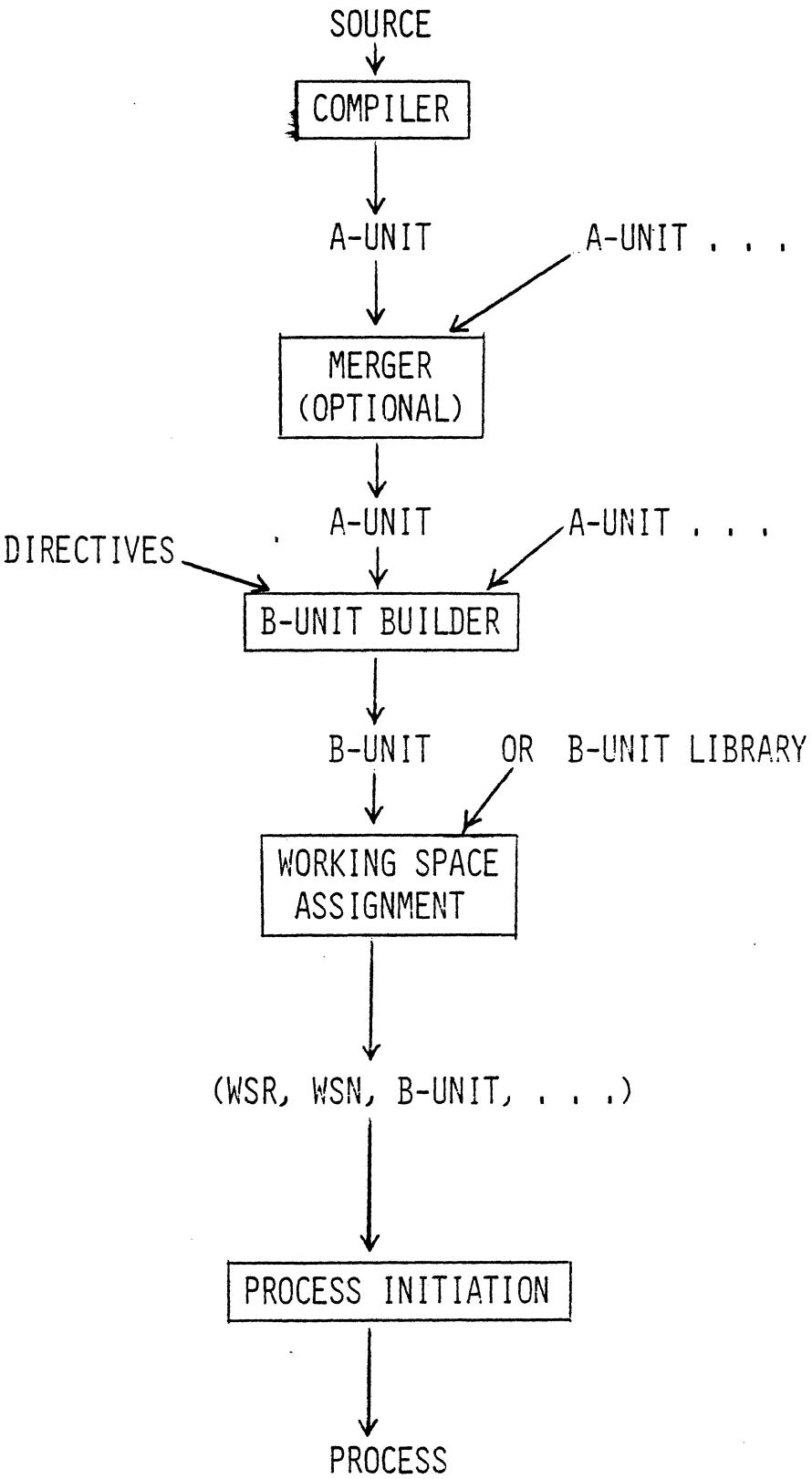
A FILE OF OBJECT TEXT PRODUCED BY COMPILERS & ASSEMBLERS

DEF - B-UNIT

A FILE THAT CONTAINS A WORKING SPACE IMAGE

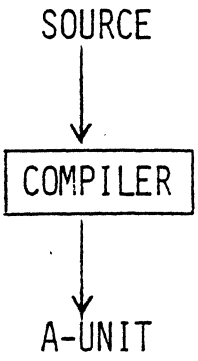
- o ONE OR MORE DOMAINS
- o LINKAGE, PROCEDURE, DATA FOR THOSE DOMAINS
- o SKELETAL PAGE TABLE

CONSTRUCTION OF VIRTUAL ENVIRONMENT



4/18/80

CONSTRUCTION OF VIRTUAL ENVIRONMENT (CON'T)



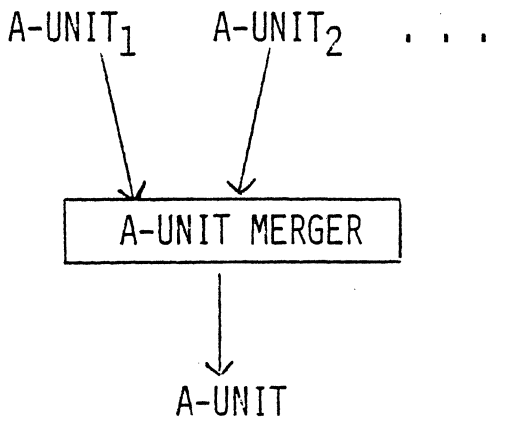
JCL

```
$ COBOL
$ PRMFL     SOURCE
$ PRMFL     A-UNIT
```

FUNCTIONS

- COMPILES OR ASSEMBLES SOURCE
- DEFINE INITIAL SEGMENT CONTENTS
- SUPPLY RELOCATION INFORMATION
- SUPPLY DEBUG SCHEMA

CONSTRUCTION OF VIRTUAL ENVIRONMENT (CON'T)



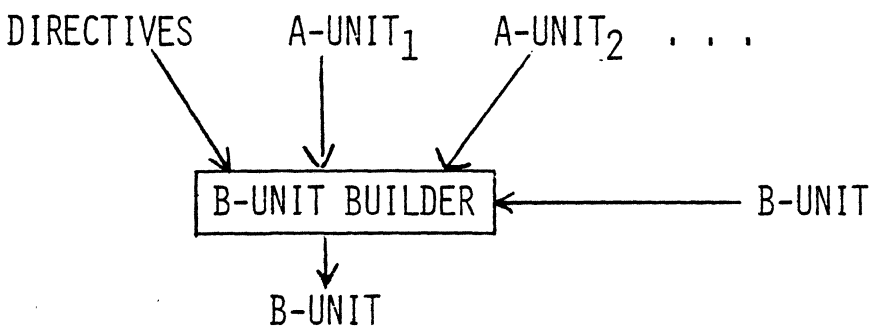
JCL

```
$ A-MERGE
$ PRMFL   A-UNIT1
$ PRMFL   A-UNIT2
    .
    .
    .
$ PRMFL OUTPUT A-UNIT
```

FUNCTIONS

- COMBINES SEGMENTS
- PERFORMS RELOCATION
- ADJUSTS SYMBOLIC SEGMENT REFERENCES

CONSTRUCTION OF VIRTUAL ENVIRONMENT (CON'T)



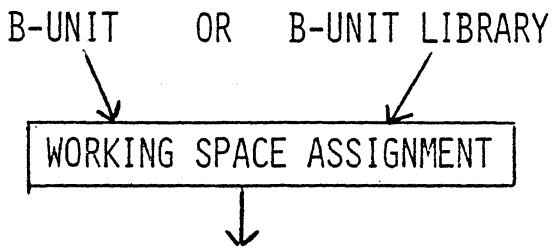
JCL

```
$ B-BUILD
$ PRMFL A-UNIT1
$ PRMFL A-UNIT2
      .
      .
      .
$ DATA
   DIRECTIVES
$ ENDCOPY
```

FUNCTIONS

- CREATES DOMAINS
- ASSIGNS VIRTUAL SPACE
- RESOLVES REFERENCES
- CREATES DIRECTORY OF DOMAINS AND GLOBAL SEGMENTS
- ADD, DELETE, OR REPLACE A-UNITS IN AN EXISTING B-UNIT

CONSTRUCTION OF VIRTUAL ENVIRONMENT (CON'T)



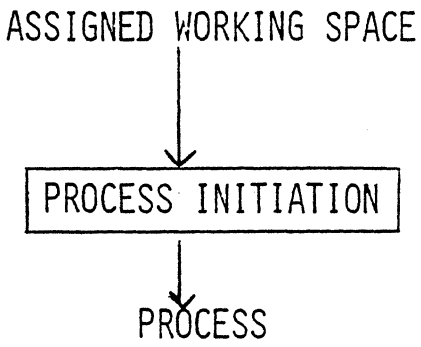
JCL

```
$ RUN
$ PRMFL   B-UNIT
$ SHRNM  SHARE LEVEL, B-UNIT LIBRARY
```

FUNCTIONS

- ASSIGNS WSR AND WSN
- CREATES A BACKING STORE FILE
- CREATES A DIRECTORY OF DOMAIN AND SEGMENT NAMES FOR ALL B-UNITS IN WS

CONSTRUCTION OF VIRTUAL ENVIRONMENT (CON'T)



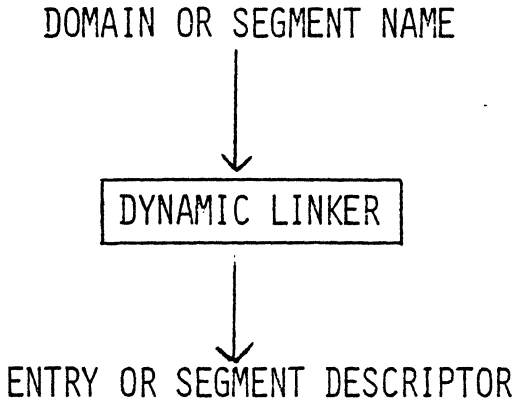
JCL

```
$ RUN
$ PRMFL   B-UNIT
```

FUNCTIONS

- ASSIGNS KPX
- BUILDS PROCESS STRUCTURE
- LOADS WSR's
- CLIMB's TO INITIAL ENTRY POINT
  (GENERATES DYNAMIC LINKING FAULT)

CONSTRUCTION OF VIRTUAL ENVIRONMENT (CON'T)



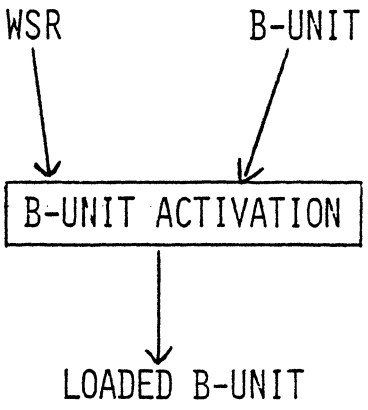
INVOCATION

- DYNAMIC LINKING FAULT REFERENCING A DOMAIN
- REFERENCE TO A SEGMENT EXTERNAL TO THE B-UNIT

FUNCTIONS

- USE SEARCH RULES TO DETERMINE THE ORDER OF WSRs TO SEARCH
- SEARCH DIRECTORY OF DOMAIN AND SEGMENT NAMES FOR EACH WSR
- IF B-UNIT CONTAINING DESIRED DOMAIN OR SEGMENT NAME HAS NOT BEEN LOADED, THEN ACTIVATE B-UNIT

CONSTRUCTION OF VIRTUAL ENVIRONMENT (CON'T)



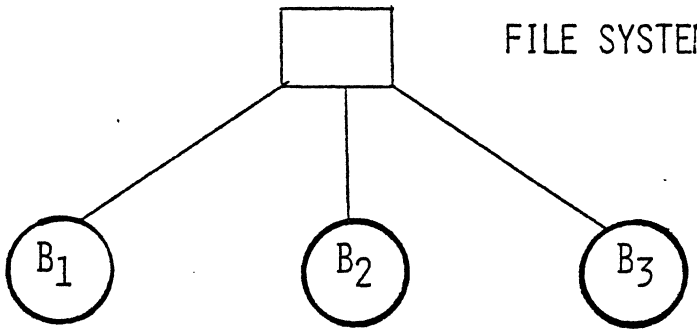
INVOCATION

- FROM DYNAMIC LINKER

FUNCTIONS

- FIX WSR VALUES IN ALL DESCRIPTORS
- IF NOT FIRST B-UNIT IN WS, THEN RELOCATE VIRTUAL ADDRESSES
- INITIALIZE B-UNIT ON BACKING STORE FILE
- ACQUIRE REAL MEMORY WORKING SET
- RESOLVE SEGREF'S VIA DYNAMIC LINKER

B-UNIT LIBRARY



DOMAINS	DOMAINS	DOMAINS
D_1	D_3	D_4
D_2		D_5
		D_6
SEGMENTS	**SEGMENTS**	**SEGMENTS**
S_1	-	S_2
		S_3

AFTER ASSIGNMENT OF LIBRARY TO WORKING SPACE:

DOMAIN DIRECTORY	
D_1	B_1
D_2	B_1
D_3	B_2
D_4	B_3
D_5	B_3
D_6	B_3

SEGMENT DIRECTORY	
S_1	B_1
S_2	B_3
S_3	B_3

4/18/80

DYNAMIC LINKING

. . .

WSR2

DOMAIN DIRECTORY

D_1	B_1	—
D_2	B_1	—
D_3	B_2	E.D.
D_4	B_3	E.D.
D_5	B_3	E.D.
D_6	B_3	E.D.

SEGMENT DIRECTORY

S_1	B_1	—
S_2	B_3	S.D.
S_3	B_3	S.D.

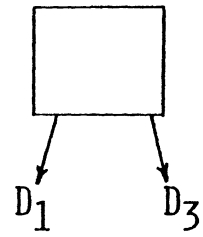
. . .

WSR7

DOMAIN DIRECTORY

X	B	ENTRY

LS (X)



4/18/80

MACRO-STRUCTURE - FUTURE WORK

- o DEFINE FORMATS FOR A-UNIT, B-UNIT
- o DEFINE WORKING SPACE FORMAT
- o DEFINE SEARCH RULES FOR DYNAMIC LINKING
- o DEFINE REQUIRED JCL
- o DETAIL SHARING CONTROL MECHANISMS
- o COMPLETE SPECIFICATIONS FOR:
  - A-UNIT MERGER
  - B-UNIT BUILDER
  - B-UNIT MERGER
  - B-UNIT ACTIVATOR
  - PROCESS INITIATION
- o SPECIFY DYNAMIC LINKING MECHANISMS
- o SPECIFY DYNAMIC LOADING MECHANISMS
- o DEFINE SYSTEM TABLES AND DIRECTORIES REQUIRED TO SUPPORT SHARING & LINKING MECHANISMS
- o SPECIFY USE OF FILE SYSTEM FOR LIBRARIES
- o TASKING

INTERNAL ENVIRONMENT

ADDRESS VALUE REPRESENTATION

- IMPORTANCE
- OPTIONS CONSIDERED
- COMPARISON
- CONCLUSION

INTERNAL STRUCTURES

- SOFTWARE STACK
- PROCEDURE SEGMENT LAYOUT

PERFORMANCE

- NON-ADP
- ADP

FUTURE WORK

REPRESENTATION OF ADDRESS VALUE IS IMPORTANT BECAUSE:

HIGH ORDER LANGUAGES USE ADDRESS VALUES FOR:

POINTERS	EXCEPTION PROCESSING
ENTRIES	STACK CONTROL INFO
LABELS	PARAMETER REFERENCING
ALTERNATE RETURN	ARGUMENT LIST BUILDING
BASED STORAGE	LOCATE MODE I/O
	OUTER BLOCK REFERENCING
	CONNECTION TO RUNTIME

BECAUSE THESE FACILITIES ARE WIDELY USED, THEY MUST BE IMPLEMENTED EFFICIENTLY, BE EASY TO USE, AND MUST BE SUFFICIENTLY POWERFUL TO SUPPORT MULTIPLE HOL USE.

ADDRESS VALUE OPTIONS CONSIDERED

		REQUIRES
1. NSA POINTER	BIT ADDR | SEG ID	LINKAGE SEGMENT
2. DESCRIPTOR + AR VALUE	BYTE BDRY | FLAGS BASE BIT ADDR |	DESCRIPTOR SEG
3. TABLE INDEX	BIT ADDR | SEG INDEX |	"CANONICAL" DS +1 ODR
4. SUPER POINTER	BIT ADDR | EXTENDED BASE	"CANONICAL" SUPER DESCRIPTOR +1 ODR

4/18/80

EVALUATION OF ADDRESS VALUE REPRESENTATION

CRITERIA	NSA PTR	DESC.	TABLE	SUPER
1. STORABLE IN DATA SPACE	Y	N	Y	Y
2. UNIFORM REFERENCE TO PARAMETERS INDEPENDENT OF DOMAIN PACKAGING	Y	Y	Y	N
3. RETAIN IDENTITY WHILE IN ODR	Y	Y	N	Y
4. SUPPORTS SEGMENT-LEVEL PROTECTION	Y	Y	Y	N
5. CAN ADDRESS > 256K	N*	Y	Y	Y
6. CAN HAVE > 1024 SEGMENTS	N*	Y	Y	Y
7. RELATIVE HIGH PERFORMANCE	N?	Y	N	Y
8. VALID ACROSS DOMAINS	N	Y	N	N
9. BIT LEVEL ADDRESSABILITY	Y	Y	Y	Y

---------ABOVE THIS LINE, N IS UNACCEPTABLE

* CAN BE IMPROVED BY HARDWARE CHANGE

? SOME HARDWARE IMPROVEMENT POSSIBLE

4/18/80

CONCLUSIONS ON ADDRESS VALUE REPRESENTATION

CAN'T USE DESCRIPTOR + OFFSET

- NOT STORABLE IN DATA SPACE

CAN'T USE TABLE INDEX

- LOADING TO ODR LOSES SEGMENT NUMBER
- REQUIRES SEVERAL INSTRUCTIONS TO LOAD

CAN'T USE SUPER POINTER

- NO WAY A PROCEDURE CAN TELL WHETHER TO REFERENCE PARAMETERS RELATIVE TO THE SUPER DESCRIPTOR FOR THE DOMAIN OR RELATIVE TO THE PARAMETER STACK
- COMPROMISES INTRA-DOMAIN SEGMENT PROTECTION, SINCE ALL DATA REFERENCE IS THROUGH ONE DESCRIPTOR

ONLY ALTERNATIVE LEFT IS NSA POINTER

- DESPITE PERFORMANCE PROBLEM

SOFTWARE STACK

EACH DOMAIN HAS A SOFTWARE STACK TO CONTROL INTRA-DOMAIN TRANSFERS AND EXCEPTION PROCESSING

ROOT FRAME

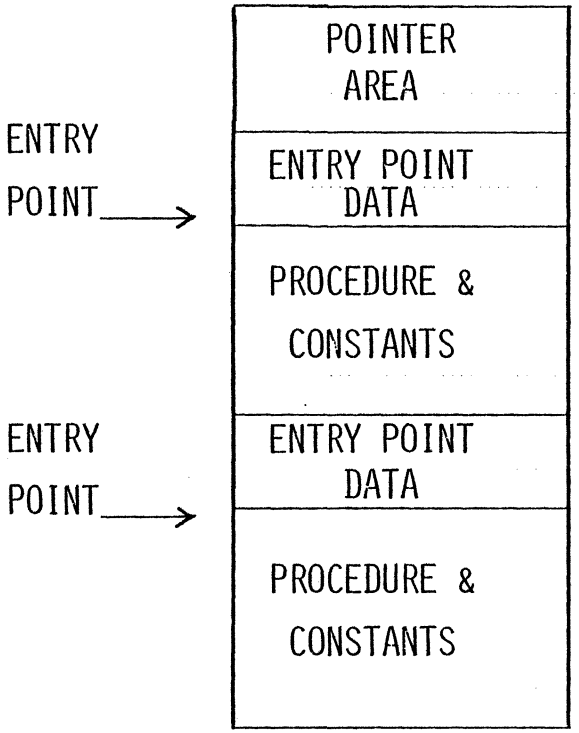
- CREATED ON DOMAIN ENTRY
- POINTS TO EXCEPTION PROCESSING ARRAY
- CONTROLS STACK SPACE
- UPDATED DURING EVERY CALL

BASIC FRAME

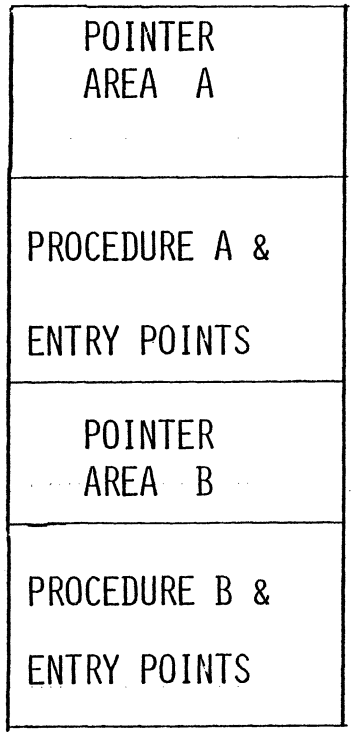
- REGISTER SAFE STORE
- PARAMETER HANDLING
- AUTOMATIC STORAGE SPACE

PROCEDURE SEGMENT LAYOUT

SINGLE PROCEDURE SEGMENT



MERGED PROCEDURE SEGMENTS



4/18/80

FUTURE WORK

STACK FRAME FORMAT & CONTENT
STANDARD CALLING SEQUENCE
ARGUMENT LIST FORMAT
EXTERNAL ENTRYPOINT CONVENTIONS
ADDRESSING CAPABILITIES WITHIN OBJECT UNIT
PLS VS. CANONICALIZING OF LINKAGE SEGMENT
HOW A PROCEDURE FINDS ITS LINKAGE
HANDLING OF LARGE ARRAYS
EXCEPTION HANDLING
SUPPORT OF ON UNITS AND SIGNALLING
SEGMENT LEVEL SHARING
OPERATOR SEGMENT ADDRESSING, SHARING, LOCATION
RUNTIME SYMBOL TABLE & DATA DESCRIPTION SCHEMA
DYNAMIC LINKING SUPPORT
I/O SYSTEM INTERFACE
TASKING SUPPORT
CALL/CANCEL SUPPORT
PRIORITY SEGMENTATION

SUMMARY

1. WE HAVE EXHAUSTIVELY INVESTIGATED THE USE OF THE NSA POINTER AS THE ADDRESS VALUE.

2. PERFORMANCE MAY BE A PROBLEM.

3. A UNIFORM SYSTEM MICRO-STRUCTURE (CALLING SEQUENCES,ETC.) IS BEING INVESTIGATED BASED ON THE USE OF NSA POINTERS.

4. A FIRST CUT HAS BEEN MADE OF THE DEFINITION OF SYSTEM MACRO-STRUCTURE (JCL, ETC.)

5. MUCH DETAILED WORK REMAINS FOR BOTH AREAS.