LEVEL 68

STANDARDS
SYSTEM DESIGNERS' NOTEBOOK

SUBJECT

Description of the Standards, Conventions, and Guidelines Used in the Software and Documentation of the Multics Operating System

SOFTWARE SUPPORTED

Multics Software Release 8.0

ORDER NUMBER

AN82-00

June 1980

Honeywell

PREFACE

Multics System Designers' Notebooks (SDNs) are intended for use by Multics system maintenance personnel, development personnel, and others who are thoroughly familiar with Multics internal system operation, and also can be used by application programmers or subsystem writers.

Since internal interfaces are added, deleted, and modified as design improvements are introduced, Honeywell does not ensure that the internal functions and internal module interfaces will remain compatible with previous versions.

This SDN describes the standards, conventions, and guidelines used in the development of all Multics software and documentation.

Throughout this manual, the term Multics refers to the Multics Operating System.

It is important that the standards in this manual be followed to consistently maintain and support the development of the Multics Operating System. If you are a system programmer, permission should be obtained from your Technical advisor to deviate from the standards; if you are not working for Multics System Development, you can deviate from the standards.

CONTENTS

CONTENTS (cont)

TABLES

SECTION 1

INTRODUCTION

HOW TO USE THIS DOCUMENT

This document should be read in its entirety in order to understand its contents. It should then be used as a reference document whenever relevant to the work being done.

VOLATILITY OF CONTENTS

The information in this document will be updated. Because the standards change as the system evolves, much of the system will not completely conform to the standards. As part of the development effort, software should be upgraded to meet the requirements of the new or changed standards whenever this is possible.

GENERAL ISSUES

To expound upon the general issues that comprise the design goals of the Multics Operating System is beyond the scope of this document. However, it is useful to remind the reader of some of the most important issues. Multics interfaces are designed with the average user in mind. As a result the needs of the very inexperienced or the very sophisticated user may be slighted. Consistency among all parts of the system is stressed in order to make this very complicated system possible to use and to allow different subsystems to be combined in ways not originally planned.

The issue of general style has always been considered important. For example, minimal use of jargon and the use of correct English is considered to be very important for both the documentation and the system interfaces themselves. Thus, diagnostic messages are sentences that begin with an upper case letter and end with a period.

REGISTERED NAMES

One of the most important parts of the user interface that must be kept consistent is that of the several classes of names used by commands and subroutines. This manual contains a number of appendices that register these names. Only those names registered here may be used; if new names are required they must be registered.

Appendix A lists control arguments.
Appendix B lists suffixes.
Appendix C lists I/O switch names.
Appendix D lists condition names.

TOPICS

Main topics included in this manual are interface standards (including both commands and subroutines), programming standards, modularity, environment independence, program format, include file format, PL/I language conventions, storage management, use of the storage system, documentation standards, info segments, and rules for translator writers.

SECTION 2

INTERFACE STANDARDS

COMMAND INTERFACES

Commands are a special class of programs designed with the terminal user in mind. They serve as the principal interface between the system and all of its users. Since many commands are used by both naive and sophisticated users, they must be designed with a two-fold purpose. The naive user should not be burdened with information he doesn't understand; the sophisticated user must not be denied the facilities that he needs.

Commands should not be too powerful. The result of a typing error should not be a disaster for the user.

Commands should be recursive, i.e., they should be able to be interrupted in midstream and invoked again. If it is inappropriate for a particular command to be used in this way, protection should be built into the command to query or inform the user.

Commands should not retain information from one invocation to the next in such a way that behavior of the second invocation is affected by some hidden implication of the first. For example, the use of the -brief control argument in one invocation should not make all future output be in -brief form. On the other hand, a metering command that accumulates data should have a -reset option to reset the accumulated values.

Storage System Conventions

In general, commands should deal with multisegment files as well as segments, unless inappropriate.

Commands should not perform write operations on the components of archive segments, except for the archive, archive_sort, and reorder_archive commands. Commands should have no knowledge of the structure of archive segments, except for the commands mentioned above and the bind command.

Generally, commands should chase links, i.e., they should perform the operation on the storage system entry to which a link points. However, there are two exceptions. Links should not be chased when the command can manipulate the attributes of the link, for example, the rename command. Links should also not be chased by default when a starname is given. The -chase control argument should be used to explicitly cause chasing of links. Refer to the copy command in the MPM Commands and Active Functions, Order No. AG92.

A command name generally starts with a verb. Multiword names should have each word separated with an underscore. A command name should consist only of

lowercase letters and the underscore character. The pl1 command is an obvious exception, but doesn't invalidate the rule. Its name was chosen by another guideline for constructing names.

A command name should be short enough so that it may be conveniently typed. A two to four letter abbreviation should be selected using the first letter of the words or syllables of the name. If the command is in the tools library, it should not be supplied with an abbreviation. The system libraries should be checked to ensure that there are no conflicts with other installed command names and abbreviations.

New command names should be chosen with the following distinction in mind. Commands whose names start with print should have no knowledge of the structure of the input and deal with ASCII data. Commands whose names start with display should know about the data structure of the input and produce highly structured output. Commands whose names start with dump should have little knowledge of the contents of the data, may produce octal output, and may format the output in blocks.

Commands that are part of a subsystem should have the subsystem name or an abbreviation as a leading name.

In general, the suffix command should be the principal command that processes a segment whose name ends in .suffix. This is particularly true for language translators, for example, the pl1 command.

Command Arguments

Commands should check carefully for argument validity and warn users of possible misunderstandings. They should be consistent in behavior and interface with other commands.

Commands that always require arguments should print a usage line when invoked with no arguments. Commands that require no arguments should print a usage line if arguments were given.

Command arguments should be order independent unless the order dependency serves a useful purpose (e.g., -update of the bind command).

Commands whose interface is simple (such as the add_name command) should accept multiple arguments if appropriate.

Commands that deal with segments whose names have a fixed suffix should not force the user to type the suffix; however, they should permit the user to include the suffix as part of the name. The expand_pathname_$add_suffix subroutine performs suffix handling for such commands.

A series of related commands should have a similar or parallel syntax and control arguments.

Arguments for which sensible defaults exist should be allowed to be omitted, i.e., the last access name to set_acl and the working directory as a default pathname to many commands.

It is desirable to check the validity of all arguments before beginning its execution or before terminating the command.

Command arguments should be used, whenever appropriate, to inform the command what to do. The user should not have to type additional lines of input to control the behavior of the command. For cases where there is a great deal of input it should be possible to put the input in a control segment and to specify the pathname of the control segment as an argument to the command.

The use of octal numbers is discouraged. An exception is the use of octal numbers to represent a segment number or segment offset, and in such cases it may be more appropriate to use a virtual pointer (as interpreted by the cv_ptr_ subroutine) or a virtual entry (as interpreted by the cv_entry_ subroutine).

Pathname Conventions

Commands that accept pathnames as arguments should accept relative pathnames or absolute pathnames as interpreted by the procedures expand_pathname_ and absolute_pathname_.

Commands that accept pathnames should honor the star convention whenever appropriate. Careful consideration of the interaction between the star convention and links must be given. Links are not chased by default when given a starname as an input argument. A control argument (-chase) should cause links to be chased when doing star processing (e.g., the copy command).

Commands that accept pairs of pathnames should honor the equal convention, for example, the compare_ascii command.

Commands that take segment names as arguments should accept pathnames and not reference names unless the command is dealing with reference names explicitly, such as the where command.

The -name control argument should be used by commands that normally take a segment number as an argument to indicate that the argument looks like a segment number but really is a pathname. The -pathname control argument should be used by commands that normally take a pathname to distinguish pathnames beginning with a minus (-) from control arguments.

Control Argument Conventions

Commands should accept control arguments that are appropriately named and that specify needed options. Similar commands should accept the same control arguments, for example, the copy and move commands. Commands that produce output should accept the control arguments -brief, -no_header, -long, and -totals, if appropriate.

Invalid or inappropriate control arguments should be diagnosed and the operation of the command terminated.

Entry names beginning with minus signs should be considered invalid by most commands. Any unrecognized string beginning with a minus sign should be diagnosed as an invalid control argument, rather than being treated as an entry name (even by commands that do not take control arguments). If there is a need to accept entry names with a leading minus sign, the -pathname control argument may be used.

New control arguments should not be invented when there are existing ones that serve the purpose. All control arguments should be registered. (See Appendix A for a list of registered control arguments.)

Control arguments that are used only by commands in the tools library should not have abbreviations.

Control arguments that take arguments should do so uniformly (from one command to another) and should always take the same type of argument.

Output Conventions

Unless the purpose of a command is to produce output, it should not produce terminal output during its normal operation. The success of a command doing its job is indicated by the absence of output. However, a command that takes a long time to execute should print a short message to reassure the user that it has started, for example, the pl1 command.

Normal output from a command should be written to the I/O switch user_output.

Error output by a command should be printed by the com_err_ subroutine. The long name of the command should be included in the com_err_ message. The com_err_ message should give additional information to resolve possible ambiguity about the source of the error. The user's argument that is in error should always be included in the call to com_err_. Calls to com_err_ should include the appropriate nonzero status code. The subroutine active_fnc_err_ should be used in a similar fashion for error reporting by active function procedures.

Error messages about storage system entries should include the absolute pathname of the entry in error. Those messages concerning input arguments should include the argument as given by the user.

Frivolous use of com_err_ should be avoided. It should not be used unless an error has actually occurred, as the condition command_error is signalled.

Other output should be done by means of iox_$put_chars or ioa_.

Red shift should not be used, since few terminals-handle this mode properly. Messages with underlining should be avoided.

Commands that produce a large amount of output (e.g., the PL/I compiler issuing diagnostics) should write all the messages on user_output and write a single error message with com_err_ indicating that there were errors.

Commands should not write terminal-directed output to I/O switches other than user_output except when specified by control arguments.

Commands that produce only one line of output should not emit blank lines.

User Interaction Conventions

When a command that interacts with the typist produces an error message that the typist does not expect, it should perform a resetread operation on user_input so that the user can modify his subsequent input.

Commands normally should not handle the quit condition.

Commands that interact with the typist should be prepared to handle the program_interrupt condition signalled by the program_interrupt command. This is particularly important for commands that produce a large amount of output that the user might want to abort.

Commands that ask questions should do so by means of the subroutine command_query_ so that the command_question condition is signalled.

SUBROUTINE INTERFACE STANDARDS

Subroutine Names

Names of subroutines or data bases should be descriptive of the function they perform. The name should not be so long as to be inconvenient to use.

Names of segments created by system programs that are to remain as segment names in a directory should end in underscore, to avoid naming conflicts with user-defined names.

Names should consist only of lowercase letters, the underscore character, and the dollar sign. Subroutine names which will appear as names on segments in the system libraries should end in an underscore to avoid conflicts with user program names.

Total length of subroutine names of the form a$b should not exceed 32 characters; the entry point portion of such a name should not end in an underscore.

Names of subroutines or data bases belonging to a subsystem should start with the name of the subsystem or a consistent abbreviation.

Standard status codes should always be returned if the subroutine returns an error code. Either codes in the system data base error_table_ or a registered subsystem error table should be used.

The status code should be the last argument to a subroutine.

Arguments in a calling sequence should be grouped: Input, Input, ..., Output, Output. Arguments that are both input and output should be avoided.

Arguments that are used for many subroutines should be declared in the same way for all such uses.

Subroutines that return more than one argument or that have side effects should not be called as functions.

The type of each argument should be chosen with the use and meaning of the argument in mind.

Subroutines should not check the number or type of input arguments, but assume they have been called correctly. Subroutines should not validate the correctness of their input arguments, unless it is part of their intended operation. However, subroutines which accept structure arguments should check the input structure version number for validity.

Character string arguments should be passed instead of a ptr to a string and its length. They should be declared unaligned and usually declared char(*) unless the length is fixed.

SECTION 3

GENERAL PROGRAMMING STANDARDS

This section documents conventions (to be used by system programs) that are of a general nature and do not fall within the province of the other sections of this document.

COMMAND STANDARDS

Commands should not call other commands.

Commands that perform general purpose service functions used by other commands or subsystems should be modularized into a command and a subroutine interface that implement the service function. The subroutine usually should not print messages but should return a standard status code to indicate an error condition.

Commands that accept arguments should accept a variable number of arguments and should not explicitly declare a fixed number of arguments. The subroutine cu_$arg_ptr should be used to obtain each argument and the subroutine cu_$arg_count should be used to obtain the number of arguments.

Commands should invoke standard approved general purpose subroutines and internal interfaces developed specifically for the command or subsystem. They should not call internal interfaces of other commands or subsystems.

Commands that are also active functions should print the results that would be returned by the active function.

Active functions that would be meaningful commands should also be invokable as commands.

NAMING STANDARDS

Programs that produce output in a file should name the output file in a sensible fashion. Listing output from the processing of segment xxx.suffix should be placed in xxx.list.

The names of all new condition names should end in an underscore to avoid naming conflicts with user-defined names. Unless an I/O switch name is an input argument, the names of all I/O switches created for use by the program should identify the program subsystem to provide meaningful output from print_attach_table. Also, the names should include a unique string to differentiate switches used by recursive invocations of the program or subsystem.

Programs that require temporary segments should use the subroutines get_temp_segment_ and get_temp_segments_ to obtain them and should call release_temp_segment_ or release_temp_segments_ when they are finished.

Programs should terminate any reference names that they associate with a segment. Programs generally should initiate a segment with a null reference name.

System programs should not allocate storage in the user free storage area. Storage is allocated in the user free area unless an <in option> is given in the PL/I <allocate statement> and <free statement>. Therefore, <in option>s must be used. Such programs should use the area obtained by calling get_system_free_area_.

Programs should have a cleanup condition handler to free allocated storage and temporary segments.

The duplicate name convention implemented by the subroutine nd_handler_ should be honored by user ring programs.

Programs should use the subroutine delete_ for deleting storage system entries. If a nonzero error code is returned, the subroutine dl_handler_ should be used to resolve the error. Programs should use the subroutine term_ for terminating segments that may have nonnull reference names associated with them or that may have links snapped to them.

Output Conventions

Programs other than commands generally should not produce output or print error messages except for those subroutines whose explicit purpose is to perform these functions. The exact nature of the output should be documented in the interface description.

If a subroutine prints an error message, it should use the com_err_ subroutine (if printing a fatal error on the caller's behalf where the caller has too little information to print a meaningful message), or the sub_err_ subroutine (if printing a nonfatal error which can be restarted selectively under control of the caller or the user). The name of the command invoking the subroutine should be given in such error messages, implying that this name must be an input argument to the subroutine.

Use of On Units for the Cleanup Condition

The on unit for the cleanup condition must not be a goto statement, and the on unit (or any procedure or block invoked from the on unit) must not contain a nonlocal goto statement, as this will interfere with the nonlocal transfer that originally signalled the condition.

Cleanup tasks include freeing allocated storage and releasing temporary segments.

Cleanup handlers should never print anything.

Programs should only be written in the PL/I language, except with explicit permission. Exceptions usually are made when performance is an issue.

All variables should be declared.

All parameter lists for external entries should be fully declared, except for those entries accepting a variable number of arguments. These should be declared as options (variable).

Declarations should be grouped in a readable fashion.

All pad fields in data structures should be explicitly declared, i.e., there should be no gaps.

All pad fields that are documented to be zero should be set to zero explicitly. This operation allows for future expansion of a data item. A name such as mbz1 should be used to declare pad fields to make it clear that the caller must zero such fields.

Whenever possible, avoid including system parameters as constants in a program. For example, if the maximum number of words for a segment is required, the external variable sys_info$max_seg_size should be referenced.

Programs should not use external state variables to pass data values between external programs. This obscures the operation of such programs. Instead, the data should be passed as arguments.

SECTION 4

MODULARITY

The Multics Operating System is modularized to simplify debugging and modification and to increase reliability. In the following section, the issues and modularity which are important to designers and implementors are discussed.

PROGRAM STRUCTURE

Overall structure of each program component is to be designed with the newest techniques. Small parts of programs are to be as understandable as the total program. Using PL/I internal procedures is encouraged because they are almost as efficient as in-line code.

COMPILABLE UNIT SIZE

Programs are recommended to be no longer than a few hundred lines of executable code. Try not to combine small components into one very large program, unless performance is affected.

GENERALITY OF MECHANISM

Much of the uniformity of mechanism and function in the Multics system has been attained by attempting to generalize mechanisms so that they can be shared. This eliminates the need for similar facilities in distinct parts of the system that are likely to produce incompatible effects or later require unwieldy simultaneous extension. Do not generalize a mechanism if it is not justifiable, however.

EXTERNAL AVAILABILITY OF MECHANISM

Make each designed mechanism externally available to the rest of the system only if it is useful and there is a demand for it. Making a mechanism externally available has implications to future changes, as compatibility must then be maintained. As a result, the designer should lean towards not making it available.

The individual program components of the Multics system are usually bound together by the Multics binder. This is done to reduce the number of separate components visible in the system, to make internal interfaces unavailable, to conserve segment numbers, and to minimize the number of linkage faults.

Contents of Bound Segments

Programs that are logically related should be bound together and placed in an order chosen to minimize the number of page faults taken when various subsets and usage patterns of the components within are used.

Names of Bound Segments

Every bound segment is named with a functionally related name starting with the string "bound_" and terminating with an underscore. The bindfile is named bound_segment_name_.bind. The bindfile and all of its components are stored in a single archive segment whose name is bound_segment_name_.archive.

Bindfile Contents

The contents of the bindfile should abide by the following conventions:

1. The order statement is required to specify binding order.

2. An objectname statement is required to state the name of the bound segment.

3. A global: delete; statement is required to ensure that extraneous definitions are deleted.

4. An addname statement is required to specify all names that are to be known externally.

5. Statements used only for debugging purposes are not to be used (e.g., No_link, Force_Order).

6. An objectname statement should be included for each component that has other attributes specified. Components with no attributes specified should not have a corresponding objectname statement since they will already have been named in the Order statement.

7. The retain statement is to be used only for those definitions that must be retained because they are externally available interfaces.

8. The synonym keyword is to be used to specify all synonyms for each component.

Bindfile Formatting

The following formatting rules are to be used for ease of reading of the bindfile:

1. Tabs should be used to separate keywords from their arguments.

2. Each objectname statement should be preceded by one blank line.

3. Synonym, delete, retain, and global statements should be indented one space with arguments lined up under the arguments of the objectname statement.

4. Comments should be included at the beginning of the bindfile that give the logical relationship of bound components, and that state the date, author and reason for each change.

5. A comment containing the word END should be placed at the end of the bindfile.

SECTION 5

ENVIRONMENT INDEPENDENCE

Subsystems, commands, and subroutines should be implemented so that they may be used in a variety of environments without other portions of the environment affecting them and without their affecting other components of the environment. This is referred to as the principle of surroundability. Surroundability is important because it makes it possible for users and subsystem designers to integrate many portions of the Multics system to compose a new subsystem without having to change any of the components they are integrating.

REENTRANCY

Components of the Multics Operating System should be reentrant. This is accomplished by proper use of pure procedures, the recursive nature of the PL/I language and its implementation, and careful use of static storage. If making a subsystem reentrant results in performance degradation or implemention problems, it is permissible for the subsystem to not be reentrant. In this case, checks are to be implemented to detect possible misuses of the subsystem and are to either prevent the misuse or warn the user of the possible conflict.

TRANSPARENCY

The operation of a program should be transparent to the environment. This is accomplished by correct storage management techniques, programs cleaning up any temporary environment changes, naming conventions, I/O attachments, and avoiding other static effects on the environment. For example, if a program changed the working directory without that being a specified property of that program, then subsequent programs would behave differently than expected.

INTERRUPTIBILITY

Programs are to be implemented so their operation can be interrupted and resumed at a later time. Programs must take precautions while they are making critical modifications to the environment. They should either prevent interrupts or be prepared to recover properly if interrupted and not resumed until after other operations that also affect the environment have been performed. For example, the I/O system should mask IPS signals when modifying IOCBs so that the quit handler will not be invoked to write a message at a time when the I/O system will not work correctly.

CONDITION HANDLING

Programs should handle conditions that may be signalled because of their operation and pass on all other conditions that they cannot handle better than the default handler.

ACCESS ASSUMPTIONS

Programs should execute properly under a wide range of access conditions. Access checking and condition handlers can achieve this. Programs using privileged entry points for some functions, but that are also usable by the general user, should be prepared to handle a call by a user with insufficient access.

USE OF STANDARD MECHANISMS

Programs ought to use the standard mechanisms that are available in the system, not their own. This is important since these mechanisms are provided to permit surroundability and modularity. For example, if a subsystem needs to use a number of temporary segments, it should use the system-provided temporary segment manager, not any other.

PATHNAMES AND SEARCH RULES

Programs should not have pathnames built into them. The standard search rule mechanism should be used to find other programs and data bases, found through the linker. Commands or subsystems for data segments should use the standard search path facility. The default search paths for a subsystem should be chosen with the general user in mind and for a specific application.

SECTION 6

PROGRAM FORMAT

This section gives guidelines and requirements for the format of programs. Recommendations mostly apply to any source language, but a few are specific to the indicated language.

COMMENTS IN PROGRAMS

All programs installed in the Multics system must have a set of comments. These comments include a copyright notice to protect the rights of the owners and sponsors of the Multics system, a journalization notice for each installation to record changes, a set of comments describing the interfaces of the program, and comments that help the reader understand the program itself.

Copyright Notice

Each separately compilable program must have a copyright notice. The copyright notices should be created and modified by the add_copyright command to ensure that all requirements are met.

Journalization Notice

Each time a program is submitted for installation to the system it must have a comment added that summarizes the reason for the change. The minimal information required is the date, name of submitter, and a one-line summary of the change, including all relevant MCR numbers. These journalization notices are to be placed after the copyright notice with the latest notice at the end. If a program is completely rewritten, a note should be made and all previous journalization notices deleted. Similarly, very old notices that no longer serve a useful purpose can be deleted.

Interface Descriptions

Each program that is an internal interface to some portion of the Multics system should have a set of comments, written in MPM style, specifying the interface or calling sequence. (If the program is an external interface that is documented in the MPM or a PLM, no such notice is necessary. A reference to the MPM or PLM volume by name and order number should, however, be included.)

Program Comments

Each major block of the program has to contain comments describing its function. Comments describing important variables, particularly those whose value have a significant effect on the flow of control through the program, but whose names do not indicate the meaning, should be included. Statement-by-statement comments are discouraged if they give little or no additional information. For example,

```
len = 0;      /* set len to zero */
```

gives no additional information, whereas,

```
len = 0;      /* for this case we have a null string */
```

offers extra information that may aid the reader.

Good choice of variable names eliminates the need for excessive commenting. However, if the names chosen are too long, they might not be able to fit on one line and may be hard to read.

GENERAL LAYOUT OF A PL/I PROGRAM

White space in the form of new pages, tabs and blank lines should separate independent sections of code. A new page should follow:

1. journalization notices
2. local declarations
3. include files

Although the following is not a strict rule, a readable program might be formatted as follows:

1. the copyright notice,
2. a general description of the program followed by the journalization notice,
3. a new page,
4. the main procedure statement,
5. declarations for variables not in include files,
6. a new page,
7. an optional set of include files,

7a. a new page if include files were included,

8. the executable code for the main body of the program,
9. a new page,
10. internal procedures for the program, and
11. another optional set of include files.

Declaration statements are to be grouped by storage class or function, and should rarely take up more than one line of code except for structure declarations. Comments should accompany variables whose use may not be obvious. Factoring of attributes should not be done to more than one attribute at a time.

Standard Format

All PL/I programs must be formatted with the standard formatting command. Programs should be structured so that subsequent invocations of the standard formatting command, such as by installation procedures, do not destroy the format.

ALM PROGRAM CONSIDERATIONS

There are conventions which ALM programs should follow. These are:

1. all entry's and segdef's should be at the beginning of the program,
2. use blank lines to separate logical statements,
3. always use include files for references to variables in structured data bases,
4. segref should not be used (use <seg>|[offset])
5. avoid VFD pseudo-ops if ALM has a cleaner way to define the data,
6. use new pages for subroutines,
7. "declarations" (equ's, temp's etc.) should be at the beginning,
8. the "push" pseudo-op should not be used with an argument, and
9. no IC references should be made when the value is greater than 2. For example,

```
*+3
3,ic
```

are both invalid.

SECTION 7

INCLUDE FILE FORMAT AND CONSTRAINTS

INCLUDE FILE FORMAT

The include files used in the Multics Operating System should be in the following form:

1. Header lines (in comments) giving the date the include file was created, dates when it was modified, as well as why, how, and by whom. The header should begin with:

   BEGIN INCLUDE FILE xxx.incl.lang

   where xxx and lang are filled in appropriately.

2. The body of the include file giving declarations (or whatever). All declared variables must be commented. Any strange constructs should be clearly described.

3. The last line of an include file should be of the form:

   END INCLUDE FILE xxx.incl.lang

   commented appropriately.

The following constraints apply to the variables and structure of include files:

1. Structures in include files should be based.

2. If a structure in an include file is based on a particular pointer, that pointer should be declared (without explicit storage class) in the include file.

3. Include files should not contain partial PL/I statements.

4. Include files should be formatted in a manner consistent with the system standards for the language in which they are written.

USE OF INCLUDE FILES

An include file should be used whenever more than one program references structured data. Include files can also be used to guarantee identical assumptions about naming conventions and systems of encoded values. Include files should not be used to include code that can be referenced by a subroutine call.

If an include file exists that describes a given data structure, that include file should be used rather than creating a slightly different one describing the same structure.

NAMING INCLUDE FILES

The name of an include file is simply:

xxxxx.incl.lang

where xxxxx is the name used in the "include" statement and lang is the name of the language for which the include file applies. The primary name of the include file (xxxxx) should end in an underscore for externally advertised include files.

Include files used by (or supporting use of) a particular subsystem should have names beginning with a prefix which identifies the subsystem. For example:

iox_modes.incl.pl1
pl1_stack_frame.incl.pl1
mrds_users.incl.pl1

PL/I AND ALM INCLUDE FILES

When a structure or data base is described in both PL/I and ALM include files, the include files are to make reference to each other. Also, the variable names should correspond exactly.

SECTION 8

PL/I LANGUAGE CONVENTIONS

This section highlights the coding rules for system programs that are to be written in the PL/I language. Recommendations for generating efficient code are included. The rules are to be taken as guidelines; there will be rare programs that follow every rule. Refer to "Efficient PL/I Constructs" below.

PL/I is the Multics Operating System programming language. However, there are several features in the language which should be avoided either because they are inefficient, they are not implemented well by our compiler, or they lead to complex coding constructs. The following language features should be avoided by subsystem programs:

1. use of PL/I input/output statements
2. aggregate expressions (except for assignment)
3. condition prefixes
4. use of "returns (char (*))"
5. use of the built-in function decat

CONSTRAINTS

The following list describes some general restrictions and requirements:

1. all variable names have to be declared in a declare statement
2. each reference to a member of a structure must be qualified by the name of the level-one containing structure
3. no compilation warning messages or error messages are allowed
4. implicit conversions should not be used
5. multiple block closures by an end statement should not be used
6. the default statement should not be used
7. executable statements should be used to initialize automatic variables to make the action more explicit, rather than the initial attribute.
8. no variable names should be the same as keywords in the PL/I language.

EFFICIENT PL/I CONSTRUCTS

This subsection is an informal guide to efficient use of the Multics PL/I compiler. It provides advice on how to take advantage of the good features of the compiler while avoiding its weaknesses. Emphasis is placed on constructs which produce more efficient code than others. The reader is assumed to be familiar with PL/I.

For a semiformal definition of the language supported by the Multics PL/I compiler, see the Multics PL/I Language Specification, Order No. AG94.

The Alignment Attributes

The use of the aligned attribute and the unaligned attribute can have a great effect on the speed of a program and the size of its data base. Unaligned items can start on a bit boundary (character boundary for character strings, pictures, and decimal variables), aligned items must start on at least a fullword boundary and occupy an integral number of fullwords. If a value requires 72 bits or less of storage to represent it, access of the value will be faster if its generation of storage is aligned because it can be directly loaded into the aq registers.

ATTRIBUTES WITH ARITHMETIC AND POINTER VARIABLES

Access of aligned binary and pointer variables is usually faster than that of unaligned variables. The only exception to the above is that unaligned pointers that the compiler recognizes as aligned are accessed at speeds comparable to that of aligned pointers, but the former cannot be indirected through. You should use aligned binary and pointer variables for local scalar variables, and only use unaligned binary and pointer variables in large data structures where size is important, but speed of access is not.

The alignment attribute has no effect on the access time of decimal variables or varying strings.

USE OF THE ALIGNMENT ATTRIBUTES WITH SHORT STRINGS

A short string is defined to be a nonvarying string with constant extents whose length is less than or equal to 72 bits (eight characters). Access of aligned short strings is usually much faster than that of unaligned short strings. Thus, it is recommended that one use aligned short strings for local scalar variables, and restrict the use of unaligned short strings to large data structures where space is important.

USE OF THE ALIGNMENT ATTRIBUTE WITH LONG STRINGS

All nonvarying strings that are not short are considered to be long. Because these strings are too long to fit into the aq registers or their length is not known at compile time, the use of the aligned attribute does not speed up their access. It is recommended that one use the default alignment attribute--unaligned.

USE OF UNALIGNED SHORT VARIABLES IN ARRAYS AND STRUCTURES

For the purposes of this discussion, short variables are those variables which occupy no more than 72 bits (eight characters) of storage and are declared with constant extents.

When accessing an element of an array of short unaligned variables, the access code is quicker if a constant subscript is used, because the compiler uses an EIS (Extended Instruction Set) instruction, when the subscript is not constant, in accessing the variable. If an unaligned short variable is contained in an array of structures, and the variable is accessed with a nonconstant subscript, access code is faster if the array is declared aligned, because the use of an EIS instruction is avoided.

Use of the Precision Attribute in Offset and Length Expressions

Because the Multics CPU's index registers can only hold 18 bits of information, while up to 24 bits may be needed to express the offset or length of a string for use in an EIS instruction, the compiler must make use of the precision attribute in deciding which register to use. If a subscript expression, the second or third argument of the substr builtin, or the declared length of a string has a precision of 18 or less, it can be kept in an index register, whereas if the precision is more than 18, it must be kept in the a or q register. This means, for example, that if a user knows that he wants a substring that may be more than 262,143 items long, then the precision of the third argument of substr should reflect that fact (otherwise the high-order bits of the length may be lost). Conversely, if the user knows that a string is less than 262,144 items long, he should reflect that knowledge in the precision used for subscripts and arguments to substr. (Besides looking at the precision of the length and offset expressions, the compiler also makes use of the declared string size in cases of constant extents to determine where the offset or length may be kept.)

The general guideline is to always declare variables with the correct precision. The following precisions are guidelines when a user is not sure a smaller precision will suffice. A word offset into a segment should be declared fixed bin(18). A number of words on a segment should be declared fixed bin(19). Character string indexes and lengths should be declared fixed bin(21). Bit string indexes and lengths should be declared fixed bin(24).

The Use of Internal Static to Simulate Named Constants

If a variable is declared to be internal static with an initial attribute and is never set within a program, the compiler will treat it as if it were a constant. (A variable is considered set if it appears on the left side of an assignment statement, is the first argument of a pseudovariable, or a reference of a defined attribute.) Converting an internal static variable to a constant means that more efficient code will often be generated to use the variable, sometimes avoiding storage references, and that the variable will not have to be copied into the combined linkage section upon initiation of the segment. Since passing a variable as an argument is equivalent to setting it, one must enclose the variable in parentheses if it is to appear in an argument list. This will make the variable be passed by value and force a copy to be made at call time. The options(constant) attribute may be used to tell the compiler that the variable is not set even if passed as an argument. Making sure that such an internal static variable, which the user intends to use as a constant, is considered by the compiler to be a constant is worthwhile if the variable is not a long string which is only used in a few calls. This feature of the compiler is a good substitute for named constants which the PL/I language generally does not provide.

When "options(constant)" is added to the declaration of an internal static initialed variable, the variable is allocated in the text section whether or not it is set or passed as an argument. The user is responsible for ensuring that the variable is not actually set, however, as this would cause faults or other errors.

Use of the Initial Attribute

The compiler's implementation of the initial attribute for automatic, based, and controlled arrays is inefficient compared with the code the user can get from explicit assignment statements. Therefore, using the initial attribute in the above cases is discouraged. Since the use of the initial attribute does not generate code for static variables, the above statement does not apply in that case. Users are warned, however, that use of the initial attribute can make a program more difficult to read in some cases, and that initialization of large external static arrays this way can cause creation of a larger object segment than intended.

The Assignment Operation

THE MULTIPLE ASSIGNMENT STATEMENT

In deciding whether or not to use a multiple assignment statement rather than separate assignment statements, it is useful to know under which circumstances multiple assignment statements produce inefficient code. A multiple assignment statement of the form:

```
T1, T2, ---, Tn = E;
```

where E is not a constant, is semantically equivalent to the separate statements:

```
V = E;
T1 = V;
T2 = V;
   .
   .
   .
Tn = V;
```

If the temporary represented by V can be kept in a machine register throughout the assignment, then the multiple assignment statement is efficient. Clearly, this implies that if E is longer than two words, the multiple assignment statement will not be efficient, since E cannot fit in a register. Thus, multiple assignment statements are not efficient when the right hand side is a long string, a varying string, an entry value, a label value, a file value, a format value, an area, a decimal value, a complex value, or an aggregate.

CONVERSIONS

All of the PL/I conversions are efficient, many of them producing inline code, while the others produce calls to any_to_any_. Inline code is produced for all cases where neither the source nor target are complex, decimal, character string, or picture (see the discussion of pictures below). Of the other cases, the following produces inline code:

```
complex_float binary (precision≤27) =   real binary;

real binary                 =   complex_float binary (precision≤27);

real decimal                =   real decimal;

complex decimal             =   complex decimal;

real binary integer         =   real decimal;

real decimal                =   real binary integer;

character                   =   real fixed decimal;

character                   =   real binary integer;
```

All other cases produce calls to any_to_any_.

The convert builtin function can be used to effect conversion between character and binary and to avoid intermediate conversions that other builtins might cause.

PICTURES

The use of pictures provides a convenient way to get efficient controlled conversion between arithmetic and character. When using pictures, the user can avoid PL/I's inconvenient conversion rules by specifying the desired format.

While picture unpacking (going from character to arithmetic form) is done by pl1_operators_, the most common cases of picture editing (going from arithmetic to character form) are done inline. Inline code is generated for the majority of cases of editing into real fixed pictures. The cases of editing into real fixed pictures that is done by pl1_operators_ are any of the following:

- the absolute value of the number's scale is greater than 31
- a "y" picture character appears in a drifting field picture (e.g., $$$y99)
- a zero suppression character or drifting character appears to the right of the "v" picture character
- the inline sequence requires more than 63 micro-ops for the MVNE instruction

ARITHMETIC OPERATIONS

Most arithmetic operations are implemented with fast inline code. The one general exception is the power operator (e.g. **) which is sometimes implemented by pl1_operators_ or subroutine calls. Users are cautioned against using the "/" operator with fixed point operands as the PL/I precision rules may cause unexpected results. Use the divide builtin function instead.

BINARY OPERATIONS

Most binary arithmetic operations produce inline code. Multiplication of fixed binary (precision>36) numbers utilizes pl1_operators_ references, all fixed binary division invoked by the "/" operator causes references to slow pl1_operators_ routines.

The "**" operation generates pl1_operators_ calls for real operands and full subroutine calls for complex operands. If the operands are both real, and the second operand is a positive integer constant that could be represented as a fixed bin(35) value, inline code will be generated to do the power operation as repeated multiplications.

DECIMAL OPERATIONS

Most decimal arithmetic operations cause efficient inline code to be generated. The major exception is the case of one or both of the operands having a scale greater than 32 or less than -31. This case will often cause additional assignments or multiplications to be generated since the 6180 hardware only handles scales within the range -31 to 32.

If the power operator has decimal operands, a conversion to and from binary and/or a subroutine call will be generated.

String Operations

All string operations (as opposed to builtins) cause inline code to be generated. In addition, some special cases cause better than usual code to be generated.

SPECIAL CASE OF CONCATENATION

Concatenation is often used in constructing varying strings. A normal concatenation of the form:

```
a = b || c;
```

results in three (3) moves -- b and c are moved into a temporary, and the result is moved into a. However, a concatenation of the form:

```
vs = vs || c;
```

where vs is a varying string, results in just one move -- c is moved to the end of vs. The latter special case can be used to great advantage in building varying strings. Consider the following example:

```
vs = a || b || c;
```

results in four moves and perhaps some instructions to allocate temporaries, while:

```
vs = a;

vs = vs || b;

vs = vs || c;
```

results in three moves with no temporaries allocated.

OPERATIONS ON LONG STRINGS

Most statements of the form:

```
a = b <bool_op> c;

a = translate (b,...);

a = bool (b,c,<bolr>);
```

where a, b, and c are long nonvarying strings, cause code to be generated that performs the operation in a temporary and then moves the result into a. However, if a is the same length as the temporary would be, and if the compiler believes that a could not possibly overlap with b or c, then the operation will be performed directly in a and no temporary will be allocated.

In a statement of the form:

```
if a <op> b ...
```

or

```
if bool (a, b, <bolr>) ...
```

where a and b are long strings, the compiler will attempt to do the operation without allocating a temporary, by using an SZTL instruction if the value is not needed elsewhere.

AGGREGATE OPERATIONS

Most aggregate operations, other than simple assignment and the use of the string and unspec builtins and pseudovariables, are relatively inefficient in the present Multics PL/I implementation and should be avoided. By simple, assignment, we mean assignment statements of the form:

```
p -> aggregate = q -> aggregate;
```

Use of the Builtin Functions

Most of the standard PL/I builtin functions and pseudovariables are implemented efficiently in the Multics compiler. However, there are exceptions and special cases.

ARITHMETIC BUILTINS

With the exception of the divide builtin, all the arithmetic builtins cause efficient code to be generated. The divide builtin is inefficient only for some cases in which a fixed binary result is produced. If a fixed binary result is produced, a reference to a very slow pl1_operators_ divide routine is generated unless the result and both operands are unscaled with a precision less than or equal to 35.

STRING BUILTINS

Efficient inline or out-of-line code is generated for all but one string builtin, decat. Execution of the decat builtin is about 50 times slower than might be expected.

There are special cases of some of the other string builtins that cause more efficient code to be generated than is normally generated for the general case. These are:

index (<char_str>, <char1>)

index (<char_str>, <char2>)

index (reverse(<char_str>), <char1>)

index (reverse(<char_str>), <char2>)

index (reverse(<char_str>), reverse(<char2>))

search (<char1>, <char_str>)

verify (<char2>, <char_str>)

search (<char_str>, <constant>)

verify (<char_str>, <constant>)

search (reverse(<char_str>), <constant>)

verify (reverse(<char_str>), <constant>)

translate (<char_str>, <constant> [,<constant>])

before (<char_str>, <char1>)

before (<char_str>, <char2>)

after (<char_str>, <char1>)

after (<char_str>, <char2>)

ltrim (<char_str>, <constant>)

rtrim (<char_str>, <constant>)

copy (<char1_constant>, expression)

References to the mathematical builtin functions are compiled either into fast references to pl1_operators_ or into slower subroutine calls. The following math builtins are implemented in pl1_operators_ if they have real arguments:

atan	exp	sin	tand
atand	log	sind	
cos	log10	sqrt	
cosd	log2	tan	

All other cases produce subroutine calls.

The Call Statement and Function References

When a call statement or function reference is executed, in the general case, an argument list must be constructed which takes 3 + 2*number_of_arguments words. When the new procedure block is entered, a new stack frame is established by a pl1_operators_ routine that takes around 30 instructions. This is a high overhead to have when using an important feature of PL/I that is necessary for good programming practice. The Multics system PL/I compiler has two optimizations which can greatly reduce this overhead. First, it can decide that an internal procedure or begin block may share the stack frame of another block rather than obtaining its own. A block that does not obtain its own stack frame is called a "quick" block or procedure. Second, the compiler can build argument lists to quick procedures at compile time, if the arguments have constant addresses known at compile time. These two optimizations greatly reduce the cost of call statements and function references.

DETERMINING THE 'QUICKNESS' OF A BLOCK

The Multics PL/I compiler goes through a two stage process to determine which (procedure or begin) blocks can be quick, that is, which ones need not obtain stack frames. The first stage excludes blocks from being quick because of their properties. The following properties can make a block non-quick.

- it is the external procedure block
- it is an ON-unit
- it has I/O statements
- it has format statements
- it has ON, or revert statements
- it has automatic variables with expression extents
- it has an entry that is assigned to an entry variable or passed as an argument
- it has an entry with a star-extent return value
- it has an entry with a star-extent parameter that is called with the corresponding argument being an expression whose length is non-constant

- it has an entry that is referenced in the argument list of such a call after the aforementioned argument

In the second stage, the compiler uses a graph of the calls between blocks, to determine which of the remaining eligible blocks can be quick. The algorithm used in this stage is an iterative one based on the constraint that a quick block may use the stack frame of one and only one non-quick block and thus may effectively be invoked from only one non-quick block. In fact, the algorithm states that a quick block may be invoked from only one stack frame, and an invocation from a quick block is considered an invocation from its owner's stack frame.

A user can determine which blocks have been made quick by examining the symbols listing produced by the compiler. In the section marked, "STORAGE REQUIREMENTS FOR THIS PROGRAM" is a list of all the blocks in the program. If the line for a particular block contains the words, "shares stack frame of", that block is quick.

USING CONSTANT ARGUMENT LISTS

In generating a quick procedure call, the Multics PL/I compiler can often generate a constant argument list if the addresses of the arguments are known at compile time. This saves the cost of executing instructions to set up the argument list at runtime. At this time the following constraints must be satisfied for the compiler to generate a constant argument list:

- the quick procedure must contain no non-quick blocks
- the stack frame of the caller must be smaller than 16,384 words
- the arguments must be constants, expressions with operators, builtin references, function references, or automatic variables
- all automatic arguments must be allocated in the stack frame of the caller
- all automatic arguments must have constant extents
- all subscripted arguments must have constant subscripts

Using If Statements

In handling if statements containing logical operators, such as:

```
if x = 0 | p ^= null | x + 3<z
     then call a;

if z>3 & q = null & loaded
     then call b;
```

the Multics system PL/I compiler (as of MR4.0) attempts to generate code that uses the minimum number of operations to decide the result. This is a change from previous releases of the compiler that always evaluated the complete expression in the if statement. In order for this optimization to take place, the user must specify the -optimize control argument for the compilation, there must be no irreducible function references in the expression, and the expression must evaluate to a bit(1) value. Thus a user should feel free to use logical operators in if statements without worrying about their efficiency.

NOTE: no program may depend on the order of evaluation of operands in the expression of an if statement. Thus, the statement:

```
if p ^= null
& p —> q = 0 then ...
is illegal
```

Any program that depends on complete or incomplete evaluation of such an expression is in error, unless the expression contains irreducible function references, in which case complete evaluation takes place.

Optimization of Comparisons

The Multics system PL/I compiler (as of MR4.0) remembers in its abstract machine state model the most recent comparison or indicator setting operation at any particular point in time in generating object code. This enables it to remove redundant comparisons in constructs such as the following:

```
if a<b
     then call foo;
     else if a = b
          then ...
```

Other Constructs That Are Costly or Dangerous

- default statements
- multidimensional arrays with star bounds
- arrays of elements of star extents
- programs requiring a stack frame of more than 16,384 words

SECTION 9

STORAGE MANAGEMENT

USE OF THE STORAGE SYSTEM

System programs often request the storage system to perform certain actions on "objects" in the storage system hierarchy. These objects are directories, segments, and links. Multisegment files (MSFs) are implemented as a collection of specially named segments (components) in a directory whose bit count (not otherwise used for directories) reflects the number of components. Although multisegment files are not part of the storage system, some of the rules governing their use are similar to those of segments.

Functions performed by the storage system are: listing an ACL, setting a bit count, and making a segment known. These requests are made through the hcs_ gate. Sometimes less primitive interfaces are used to perform the functions. These are preferred because of the "side effects" they provide.

The hcs_ interfaces accept either a pathname (consisting of a directory pathname and an entry name) or a segment pointer. System subroutines should not use relative pathnames or reference names to identify an object in the storage system.

Pathnames

All storage system interfaces that accept pathnames also expect descriptors for the pathnames. The argument description in the declare statement for the entry should be "char (*)". (Some programs written with previous compilers in mind use "char (*) aligned".) The programmer is advised to make reference to the interface description.

Naming Conventions

Programs do not change the name of an existing storage system object. Objects are created with only one name. System programs should preserve all the names of existing entries which are manipulated. System programs should be aware of the command system's conventions concerning entry names and not violate them. Nonprinting characters, for example, are to be avoided.

System programs must be designed to be aware of the "primary name" concept and preserve it.

Working Directory Use

System programs creating new storage system objects (other than temporary segments) should place them in the user's current working directory, unless a target directory is specified.

Access Control List Management

System programs creating new segments and directories should set access control lists according to the convention listed below. If the segment is being changed in place, the program, upon exit, should leave the ACL as it was before the program was executed. Sometimes, a program that needs w access to change the contents of a segment finds the segment exists without w access to the current user. The program must change the ACL in order to change the contents of the segment and then restore the ACL to its former value upon completion. In this instance, a cleanup handler should be established to restore the ACL if execution is aborted. If the branch is being created, the ACL should be added to rather than replaced to ensure that the initial ACL entries are placed on the branch. The ACL entry for *.SysDaemon.* should be preserved. The access to be given to the user creating the branch is:

Segment Type	Access	Ring Brackets
directory	sma	7,7
object segment	re	V,V,V
data segment	rw	V,V,V

where V is the current validation level of the user. Access identifiers should not contain specific instance tags. The instance tag should be "*" except when the associated ACL entry is placed on a branch for a short period of time.

Making Segments Known and Unknown

A pointer to a segment is obtained by making the segment known. Segments should be made known by calling hcs_$initiate, hcs_$initiate_count, or hcs_$make_seg. These primitives all have the secondary effect of associating a reference name with the segment as well. This reference name must be null. Segments should be made unknown when processing of the segment is complete. The entry hcs_$terminate_noname should be used if the segment was made known with an associated null reference name. The subroutine hcs_$terminate_name is used in the rare case of removing a nonnull reference name from the name space of the process.

If snapped links point to a segment which is unknown, the subroutine term_ is used to cause the necessary links to be restored to their original unsnapped state.

The copy control switch and reserved segment number switch parameters to the initiate subroutines should be 0 except in rare cases.

Pathnames vs. Segment Pointers

Many functions of the storage system can be handled by either of two interfaces to the supervisor, one requiring an absolute pathname and the other requiring a segment pointer. The programmer is therefore faced with deciding which of these interfaces to use. The issues are based on two facts. First, the pathname interfaces are more expensive, and second, the pointer interfaces are more prone to errors in that segment numbers are reusable within a process. If a segment's contents are addressed directly, the pointer interfaces should be used since getting a pointer to the segment is required anyway. If several calls must be made to the supervisor for the same segment, pointer interfaces should be used due to their speed. If only one storage system request is made and the segment is not accessed, the pathname interface should be used to avoid the cost of making the segment known.

System programs calling storage system primitives with a pointer should do so with a pointer that points to the base of the segment.

If pointer calls are made in cleanup handlers, the validity of the pointer must be maintained by the program establishing the cleanup handler. If a segment is made unknown or deleted, a flag should be set to indicate that the pointer is invalid. (Frequently this is done by setting the pointer to null.) The cleanup handler should check the value of this flag before using the pointer to identify the segment. If this is not done, the cleanup operation may be performed on a different segment. Sometimes it may be better to use a pathname interface in a cleanup handler.

A program should keep a pointer (in internal static) to a data base which is referenced often rather than resolving a pathname each time the data base is referenced. Appropriate warnings should be given describing the problems of deleting the segment while it is in use. (Although using the pathname each time is safer, it is not foolproof.)

Multisegment Files

System programs operate correctly if given a multisegment file -- whenever reasonable. It is not expected that the source for a translator, for example, will reasonably overflow into a multisegment file, and, hence, translators need not expect multisegment files as input source segments. However, the listing output of translators will often require a multisegment file.

Standard subroutines perform actions on multisegment files rather than embedding knowledge of the structure of multisegment files throughout the system. The procedures msf_manager_, tssi_, make_msf_, copy_seg_ and delete_ allow manipulation of multisegment files.

Programs that open multisegment files must ensure that all such files are closed when through. Cleanup handlers are thus required. (Opening and closing multisegment files is analogous to making a segment known and unknown.)

System programs should not destroy the structure of a multisegment file, i.e., cause it to become nonstandard. System programs should not assume knowledge of the multisegment file format since it is internal to the system and may change.

Use of the Bit Count

Data bases must be kept consistent by system programs. Bit counts should be updated when a change to the content and size of a segment are complete. A cleanup handler should be enabled to make sure the bit count is set correctly if the program aborts.

The bit count is used for different purposes by various programs. The standard meaning of the bit count is that it defines the last meaningful bit in a segment. Nonzero bits in a segment beyond the bit count may lead to confusion. If necessary, programs that set the bit count should also truncate the segment and zero unused bits in the last word.

STORAGE ALLOCATION

Several choices are open to the programmer to obtain temporary storage. The preferred method is to assign space in the user ring stack segment by means of declaring automatic storage. This should be done if the maximum amount of storage is fixed, known at compile time, and is not likely to overflow the user's stack.

If the amount of storage is unknown until runtime and is again unlikely to cause a stack overflow, a begin block or an internal procedure invocation can be performed to allocate precisely the amount of automatic storage required.

Automatic storage is desirable because no cleanup handlers need to be established.

Internal Static Storage

Internal static is not recommended for any variables that are not static by nature. Using internal static as a replacement for named constants is acceptable in the some cases.

PL/I Areas

Another means of allocating storage is to use the area mechanism. If areas are to be used, the PL/I allocate and free statements should be used.

Areas can be declared to be automatic or a standard user-ring area segment may be used. A pointer to this area is obtainable by calling get_system_free_area_. Since a number of system programs make use of this segment, the cost is shared. Programs producing allocations in areas should free storage when it is no longer required. In addition, cleanup handlers must be provided to free the space if the program aborts abnormally.

Temporary Segments

Creation of temporary segments in the user's process directory is another way of obtaining storage. This should be done when the amount of space required will probably be large but is actually unknown. The subroutines get_temp_segment_, get_temp_segments_, release_temp_segment_, and release_temp_segments_ should be used to manage a program's temporary segments. Programs should always release such temporary segments upon completion.

All system programs that procure temporary segments should establish cleanup handlers to release these segments if execution is aborted abnormally.

SECTION 10

DOCUMENTATION STANDARDS

There are two forms of documentation for the Multics system. The primary form is published manuals distributed through the marketing organization, and must conform to Honeywell Documentation Standards. This set of documents includes the Multics Programmer's Manual, the Language Manuals, the System Designer Notebooks, the Administrator Manuals, and the Program Logic Manuals.

The second form of documentation is online information that can be read with the help command or printed. This documentation is distributed with each release of the Multics system, and must also conform to Honeywell Documentation Standards.

LOCATION OF DOCUMENTS

Every command must have a module description in standard MPM form. The placement of this module description is determined by the nature of the command. If the command is for the general user, the description is included in the MPM itself. If it is for Administrators, Operators, or Field Engineers, it is included in the manual provided for that class of user.

Every subroutine whose name and entrypoint are retained, and thus is externally available, has a subroutine module description prepared in the standard MPM format. The document location is determined on the same basis as for commands.

Each available command or subroutine has an info segment that describes its use. The info segment conforms to the info segment standard described in this document.

SECTION 11

INFO SEGMENTS

STYLE

Info segments are provided to help the user. They are not intended to offer instruction in the use of Multics. When writing an info segment, use the following guidelines:

1. Be brief, concise, and terse.
2. Minimize words; minimize white space.
3. Assume some knowledge and experience on the part of the reader.
4. Use active verbs in the present tense.
5. Provide only essential facts; reference manuals for complex detail.
6. Remember that info segments do not replace, but rather supplement the manuals.

PHYSICAL APPEARANCE

Info segments must be readable on all terminals supported by Multics. Therefore:

1. Employ a maximum line length of 71 characters.
2. Avoid tabs and needless spacing -- they slow output. Keep indentation to a minimum.
3. Avoid underlining. Underlined text is illegible on many terminals.
4. Avoid control characters; they may not be transparent when the system is accessed via certain terminals. This means that the use of 006 in info segments is discontinued and being replaced by a new convention described in this document.
5. Do not put the string "(END)" at the end.
6. Use MPM conventions regarding punctuation, capitalization, etc. (Described in this manual).

NAMING CONVENTIONS

The help command respects the star convention for segment names. Therefore, the following naming conventions should be followed:

1. XXX.info for command XXX, subroutine XXX, or topic XXX; e.g., help.info, random_.info, new_rates.info.

2. XXX.changes.info for changes to XXX; e.g., random.changes.info.

3. XXX.status.info for status of bug fixes to XXX; e.g., basic.status.info.

4. XXX.gi.info for text info segments on general information; e.g., master_directories.gi.info.

5. MRn-n.gi.info for system release summary info segments; e.g.,MR4-0.gi.info.

SYNTAX OF INFO SEGMENTS

There are three kinds of info segments: command descriptions, subroutine descriptions, and all others. Rules for the first two types are strict so that the user can search efficiently for particular items.

Title

Some rules apply to all three types of info segments. The first line in every info segment must be a brief title line, beginning with the date of last modification. This line should be appropriate for a table of contents; for command or subroutine descriptions it will give the name(s) of the program, including abbreviations.

Examples of title lines:

```
05/13/76  announcements of future changes
06/21/76  qedx, qx
06/22/76  random_
```

Paragraphs

Info segments consist of a series of paragraphs separated by two blank lines. The help command pauses at the end of every paragraph and asks, "More help?" so paragraphs should be neither too long, nor too short. Ten lines is the recommended maximum length for most paragraphs.

Sections

A section of an info segment consists of one or more paragraphs grouped under a title. The title is on the first line of the section; it ends with a colon, and is followed by the rest of the first paragraph. Subsequent paragraphs in the section have no titles, (i.e. their first line has no colon). The help command allows the user to print only a specified section of an info segment, or to obtain a list of the section titles in an info segment.

Command Descriptions

For command descriptions, the following sections must be present:

Syntax
Function

If the command has arguments or control arguments then the following sections are required:

Arguments
Control arguments

If other sections are necessary they may be provided. The following names may be used:

Access requirements
Notes
Examples

Other section names may be used only with the permission of the info segment censor.

The "Syntax" section gives a sample invocation of the command. For example:

Syntax: list_iacl_dir {path} {User_ids} {-control_args}

In the syntax line, optional arguments are enclosed in braces.

The "Function" section should be a terse one- or two-line description of what the command does. Since one mode of the help command allows the user to search for paragraphs containing a specified string, it is desirable to describe the function of the command using terms that a user may be searching for; but only if this can be done concisely.

The "Arguments" section gives a one-line description for each argument to the command. The "Control arguments" section gives a one-line description of each control argument. These section titles are plural even if only one item is described. For control arguments, default values should be indicated by the string "(default)". If an argument is too complicated to explain completely on one line, or if the command takes many interrelated control arguments, it should cross reference a longer section identified by section name at the bottom of the segment.

Subroutine Descriptions

For subroutines, the following sections are required:

Function
Syntax per-entry
Arguments per-entry

The "Syntax" section should have an example of the declaration and invocation of the subroutine in the PL/I language unless the subroutine cannot be used from PL/I.

The "Arguments" section is like the MPM description of the argument, but should be kept to one line per argument, with a reference to further info if necessary.

Multiple entry procedures have multiple Syntax and Arguments sections, one for each entry point.

Other Info Segments

For info segments describing other topics, the only explicit syntax rules are that the title be in standard form, and that paragraphs be of reasonable size. Section titles, when used, should not be fanciful. For segments like Installations.info, the section title should be the date; for pending_changes.info, it should be the date and the name of the program or function being changed. The name of info segments that contain general information should be topic.gi.info.

See info_seg_format.info and the documentation for input_info_seg and validate_info_seg_ for detailed descriptions of info segment syntax. Also, see list.info, io_call.info, and convert_date_to_binary.info for examples of command and subroutine info segments.

SECTION 12

RULES FOR TRANSLATOR WRITERS

THE COMMAND PROGRAM

The command program that invokes the translator should expect to find a source program with the suffix .lang if lang is the name of the command.

The command program should establish a cleanup handler to clean up any temporary storage being used by the translator in case of an abnormal termination.

The command program should establish an any_other handler to trap all unexpected faults. All system conditions should be passed on to the system default handler.

The command program should detect that it is being invoked reentrantly (i.e., with a suspended invocation dormant in the process) and, if the translator can not be used in this fashion, should ask the user if he wishes to continue and act accordingly.

THE OBJECT SEGMENT CREATED

Translators should create Multics standard object segments. The object segments should not make reference to any system data bases other than the stack segment header. No assumptions should be made about the call/push/return conventions and the format of a stack frame header. The first 32 words of a stack frame header are reserved for system use and should not be referenced by the object code.

An operator segment, containing code and necessary references to system data bases, should be used to guarantee compatibility from one system release to another. The operator segment should be found (by the object code) by means of the operator_pointers_ mechanism in the stack header.

As many code sequences as possible should be placed in the operator segment to minimize the size of the object segment.

The operator segment should contain all of the system dependent code used by the object segment (e.g., references to system data bases).

A standard table giving names of the operators should be provided. This table should be called lang_operator_names_ where lang is the name of the translator command.

LISTING OUTPUT

The translator should create an optional listing of the translation. This listing should be created whenever the control arguments -list, -map, -symbols, or -source are specified. The -map control argument should cause the translator to generate the "most commonly wanted" form of output.

The output listing, which must allow for multisegment files, should contain the following:

1. descriptive lines indicating when the program was compiled, by which version of the translator, and with what control arguments
2. a line-numbered listing of the source used, include file source should be distinguishable from top-level source
3. pathnames of all source segments used including include files
4. a list of externally referenced names
5. a list of operators used
6. a list of variables used giving storage attributes and where the variables were referenced
7. a table of the storage requirements used by the compiled program.

If the -list control argument is specified, the following should also be included:

1. a listing of all constants and literals used by the program
2. a listing of the source lines that generated the object code interspersed with the object listing appropriately
3. "comments" in the object listing indicating which variable is being referenced, which operator is being used, etc.
4. relocation information with each word of generated code
5. appropriate interpretation of noninstruction data (e.g., ASCII data should be printed out in ASCII as well as octal)

MISCELLANEOUS REQUIREMENTS

The translator should use the standard search path facility to locate any include files (or analogous constructs) used by the program being translated.

The subroutine tssi_ should be used to create object segments and listing files.

The ACL on an object segment should be left after translation just as it was before translation.

APPENDIX A

REGISTERED CONTROL ARGUMENTS

This section lists all approved control arguments and their abbreviations. Only these control arguments should be documented and accepted by commands. This applies to all commands whether they are standard user accessible commands or special tools for a particular class of users.

For reasons of compatibility with the past, many commands are permitted to accept control arguments that have been previously documented. An exception is made for commands that are of very limited interest that accept a large number of potentially obscure control arguments; these commands may have nonstandard control arguments. If these commands are upgraded for more general use, their control arguments will have to be modified.

If a command needs a control argument that is not registered in this list, it and its abbreviation should be defined. If the command is a special purpose tool and the control argument itself is not of general interest, no abbreviation should be defined.

Two different lists of control arguments are presented. Table A-1 consists of general purpose control arguments, which are already used by system commands and may be expected to cover most situations. System programmers should use items from this list whenever possible. Table A-2 consists of more specialized control arguments, which cover a more limited range of situations.

Table A-1. Approved Standard Control Arguments

-absentee	-as
-absolute_pathname	-absp
-access	-ac
-access_class	-acc
-access_name	-an
-account	
-acknowledge	-ack
-acl	
-address	-addr
-admin	-am
-after	-af
-alarm	-al
-all	-a
-arguments, -argument	-ag
-ascending	-asc
-ascii	
-assignments	-asm
-attributes	-attr
-author	-at
-authorization	-auth
-bcd	

-before	-be
-bit_count	-bc
-bit_count_author	-bca
-block	-bk
-branch	-br
-brief	-bf
-brief_table	-bftb
-call	
-category	-cat
-character	-ch
-chase	
-check	-ck
-comment	-com
-console_input	-ci
-copy	-cp
-copy_switch	-csw
-count	-ct
-current_length	-cl
-date	-dt
-date_time_contents_modified	-dtcm
-date_time_dumped	-dtd
-date_time_entry_modified	-dtem
-date_time_used	-dtu
-date_time_volume_dumped	-dtvd
-debug	-db
-decimal	-dc
-default	
-delete	-dl
-delimiter	-dm
-density	-den
-depth	-dh
-descending	-dsc
-destination	-ds
-device	-dv
-directory	-dr
-else	
-entry	-et
-every	-ev
-exclude	-ex
-execute	
-extend	
-field	-fl
-file	-f
-fill	-fi
-first	-ft
-force	-fc
-from	-fm
-gen_type	-gt
-header	-he
-hold	-hd
-home_dir	-hdr
-id	
-indent	-ind
-input_file	-if
-input_switch	-isw
-io_switch	-iosw
-label	-lbl
-last	-lt
-length	-ln
-level	-lev
-limit, -limits	-li
-line_length	-ll
-lines	
-link	-lk
-link_path	-lp
-list	-ls
-logical_volume	-lv
-long	-lg
-map	

-mask	
-match	
-max_length	-ml
-maxlines	
-minchars	
-minlines	
-mode	-md
-model	
-modes	
-multisegment_file	-msf
-name	-nm
-next	
-nl	
-nnl	
-no_acknowledge	-nack
-no_address	-naddr
-no_header	-nhe
-no_offset	-nofs
-no_fill	-nfi
-no_link_translation	-nlt
-no_list	-nls
-no_notify	-nnt
-no_numbers, -no_number	-nnb
-no_pagination	-npgn
-no_print	-npr
-no_quote	-nq
-no_restore	-nr
-no_totals, -no_total	-ntt
-no_update	-nud
-non_null_link	-nnlk
-notify	-nt
-number	-nb
-octal	-oc
-off	
-offset	-ofs
-on	
-optimize	-ot
-ordered_field	-ofl
-outer_module	-om
-output_file	-of
-output_switch	-osw
-owner	-ow
-page	-pg
-page_length	-pl
-parameter	-pm
-pass	
-pathname	-pn
-position	-psn
-previous	-prev
-primary	-pri
-print	-pr
-profile	-pf
-project	-pj
-query	
-queue	-q
-quit	
-quota	
-record	-rec
-records_used	-ru
-repeat	-rpt
-replace	-rp
-request	
-request_type	-rqt
-reservation	-resr
-reset	-rs
-resource	-rsc
-restart	-rt
-reverse	-rv
-revert	

-ring	-rg
-ring_brackets	-rb
-safety_switch	-ssw
-search	-srh
-section	-scn
-segment	-sm
-sender	
-severity	-sv
-short	-sh
-sort	
-source	-sc
-start	-sr
-stop	-sp
-string	-str
-subscriptrange	-subrg
-subsystem	-ss
-switch	
-symbols, -symbol	-sb
-system	-sys
-table	-tb
-tab	
-target	
-terminal_input	-ti
-terminal_type	-ttp
-then	
-time	-tm
-title	
-to	
-total, -totals	-tt
-times	
-track	-tk
-truncate	-tc
-type	-tp
-unique	-uq
-unique_id	-uid
-update	-ud
-user	
-volume	-vol
-wait	-wt
-working_dir	-wd

Table A-2. Approved Special Control Arguments

-4bit	
-7punch	-7p
-abort	
-accept	
-access_label	-albl
-action	
-age	
-append	-app
-attached	-att
-attachments	-atm
-ball	-bl
-bottom_label	-blbl
-bottom_up	-bu
-brief_header	-bfhe
-cancel	
-card	
-cc	
-change_default_auth	-cda
-change_default_project	-cdp
-change_password	-cpw
-check_ansi	
-chpass	
-cl	
-class	
-class_indentifiers	-cli
-cmf	
-cobol_switch	-cs
-collection, -coll	-col
-compile	
-complete	-comp
-comp_volume_dump_switch	-cvds
-copy_release_names	-crn
-continue	-ctu
-control	-ct
-control_arg	-ca
-convert	
-cput	
-date_time	
-day	
-day_name	
-debug_cg	
-deferred_indefinitely	-dfi
-definition	-def
-defs	
-delay	-dly
-detach	-det
-dir_mode	
-dont_free	
-dprint	-dp
-dpunch	-dpn
-ebcdic8	
-ebcdic9	
-edit	-ed
-entry_numbers	-etn
-entry_point	-ep
-exec_com	-ec
-expand	
-expanded	
-files	
-file_input	-fi
-flush	
-fnp	
-fold	
-format	-fmt

-foreground	-fg
-free	
-fw	
-generate_password	-gpw
-go	
-govern	-gv
-hex8	
-hex9	
-hyphenate	-hph
-immediate	
-in	
-incremental	
-inccost	
-inc_pf	
-inc_volume_dump_switch	-ivds
-inc_vcpu	
-in_reply_to	-irt
-initial_string	-istr
-inout	
-input_description	-ids
-interactive	-ia
-interactive_message	-im
-interrupt	-int
-invisible	-iv
-keyed	
-library	-lib
-line_numbers	-ln
-lmargin	-lm
-log	
-loop	
-long_id	-lgid
-lower_case	-lc
-mailbox	-mbx
-mcc	
-message_id	-mid
-meter, -metering	-mt
-min	
-minute	
-month	
-no_abort	
-no_canonicalize	-ncan
-no_endpage	-nep
-no_exec_com	-nec
-no_freeing	
-no_hold	-nhd
-no_interactive_message	-nim
-no_label	-nlbl
-no_log	
-no_message_id	-nmid
-no_null	
-no_orginal	-no_orig
-no_preempt	-np
-no_print_off	-nprf
-no_prompt	
-no_request_loop	-nrql
-no_stop_run	-nsr
-no_start_up	-ns
-no_subject	-nsj
-no_symbols, -no_symbol	-nsb
-no_warning	-nw
-non_edited	-ned
-nogo	
-notape	
-old_original	-old_orig
-open	
-original	-orig
-original_path	-orig_path
-out	
-output	

-output_description	-ods
-own	
-pdd	
-prefix	
-print_delay	-pr_dly
-print_edit	-pr_edit
-print_linkage_fault	
-print_new_lines	-pnl
-print_off	-prf
-process_overseer	-po
-prompt	
-proxy	
-raw	
-realt	
-relocatable	-rlc
-remove	-rm
-rename	-rn
-reply_to	-rpt
-report_reset	-rr
-request_loop	-rql
-retain	-ret
-retain_data	-retd
-return_value	-rtv
-runtime_check	-rck
-safe_optimize	-safe_ot
-save	-sv
-secondary	
-seg	
-seg_mode	
-separate_static	-ss
-set	
-set_bc	
-set_nl	
-single	-sg
-single_name	-snm
-size	
-skip	-sk
-sleep	
-sort_dir	-sd
-sort_file_size	-sfs
-source_switch	-ssw
-static	
-status	-st
-stop_proc	-spp
-stt	
-subject	-sj
-subtotal	-stt
-subtree	-subt
-sys	
-sys_id	
-sysid	
-tape	
-tape7	
-tape9	
-temp_dir	-td
-template	-tmp
-text	-tx
-time_ot	
-timers	
-top_label	-tlbl
-total_cost	
-total_mem_units	
-total_pf	
-total_vcpu	
-to_queue	-to_q
-to_request	-to_rqt
-trace	
-trace_linkage_faults	
-train	-tn

-unlabeled_common	-uc
-upper_case	-uc
-use_bc	
-use_count	-use
-use_nl	
-verbose	-vb
-warn	
-watch	
-year	
-zero_on_alloc	
-zero_on_free	
-zone	

APPENDIX B

REGISTERED SUFFIXES

This section lists all registered entryname suffixes and their general meaning. Programs should follow the conventions implied by these suffixes.

LIST OF SUFFIXES

absin	absentee input segment
absout	absentee output segment
alm	ALM source program statements
apl	APL saved workspace
archive	archive segment
basic	BASIC source program statements
bcpl	BCPL source program statements
bind	bindfile for the bind command
breaks	break segment for the debug command
cds	cds source program statements
chars	auxiliary output file produced by runoff and compose
ckrout	output segment form the check_mst command
cmdb	data model source segment for the create_mrds_dv command
cmdsm	data submodel source segment for the create_mrds_dsm command
cobol	COBOL source program statements
code	enciphered segment produced by the encode command
compin	compose source statements
compout	primary output file produced by compose
control	control file for miscellaneous commands
crl	control segment for the cross_reference command
crlout	output file of the cross_reference command
dict	dictionary segment used by dictionary commands

dir_info	directory information segment output by the save_dir_info command
dsm	output file produced by the create_mrds_dsm command
ec	segment containing exec_com lines
fortran	FORTRAN source program statements
gcos	file in GCOS Standard System Format
gdt	graphics device table source segment input to the compile_gdt command
ge	input segment to the graphics_editor command
header	header file for MST generator
incl.alm	ALM include files
incl.bcpl	BCPL include files
incl.cobol	COBOL include files
incl.fortran	FORTRAN include files
incl.lisp	LISP include files
incl.pl1	PL/I include files
info	segments formatted according to help command conventions
ld	source for the library descriptor command
lisp	LISP source program statements
list	listing file produced by a compiler assembler or binder
lister	data base used by lister commands
listform	control segment defining document format used by process_list command
listin	segment used by create_list command to input or update a lister file
map	map segment produced by binder or library_ map
mbx	ring 1 mailbox
memo	database used by memo command
mexp	mexp source program statements
motd	database for the print_motd command
ms	ring 1 message segment
order	control segment for the reorder_archive command
pfd	output file created by the profile command
pfl	listing file used by the profile command
pl1	PL/I source program statements
print	output of the library_print command
probe	break segment for the probe command

profile	user profile used by abbrev and check_info_segs
qedx	qedx macro
rd	reduction compiler source statements
runoff	runoff source statements
runout	primary output file produced by runoff
search	search directories for MST generator
symbols	symbol dictionary used by the Speedtype commands
teco	teco macro
volumes	output segment of the manage_volumes_pool command
wl	wordlist segment used by the wordlist commands

APPENDIX C

REGISTERED I/O SWITCH NAMES

This section lists the names and meanings of all I/O switches that are used by system programs.

LIST OF I/O SWITCH NAMES

audit_i/o.HHMM.T	switch used bu audit facility, where HHMM.T is the time switch was attached
debug_input	switch from which the debug command takes its input requests
debug_output	switch onto which the debug command writes its output
ec_switch_nn	switch associated with the nn switch attached by the exec_com command
error_output	switch onto which commands write their error messages
filenn	switch associated with unitnn in FORTRAN programs
fo_!uniquename	switch attached by file_output command
fo_save_!uniquename	switch used by file_output to save previous attachment
graphic_input	switch used for graphics input
graphic_output	switch used for graphics output
lib_map_	switch used by library_map command
lib_print_	switch used by library_print command
user_i/o	switch attached to interactive users primary I/O device
user_input	switch from which commands and the command processor take their input
user_output	switch onto which commands write their normal output
!uniquename.lila	switch name used by LINUS
!uniquename.rel	switch name used by LINUS
!uniquename.res	switch name used by LINUS

APPENDIX D

REGISTERED CONDITION NAMES

This section lists all registered condition names.

LIST OF CONDITION NAMES

active_function_error
alrm
any_other
area
bad_area_assignment
bad_area_format
bad_area_initialization
bad_dir_
bad_outward_call
cleanup
command_abort
command_error
command_query_error
command_question
conversion
cput
cross_ring_transfer
db_conversion
derail
endfile
endpage
error
fault_tag_1
fault_tag_3
finish
fixedoverflow
gate_error
illegal_modifier
illegal_opcode
illegal_procedure
illegal_return
io_error
ioa_error
isot_fault
key
linkage_error
listing_overflow
lockup
lot_fault
message_segment error
mme1
mme2
mme3
mme4
name
no_execute_permission
no_read_permission

```
no_write_permission
not_a_gate
not_in_call_bracket
not_in_execute_bracket
not_in_read_bracket
not_in_write_bracket
op_not_complete
out_of_bounds
overflow
page_fault_error
parity
program_interrupt
quit
record
record_quota_overflow
return_conversion_error
seg_fault_error
simfault_nnnnnn
size
stack
storage
store
stop_run
stringrange
stringsize
sub_error_
subscriptrange
sus_
timer_manager_err
trm_
transmit
truncation
unclaimed_signal
undefinedfile
underflow
unwinder_error
zerodivide
```

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE: LEVEL 68
STANDARDS SYSTEM DESIGNERS' NOTEBOOK

ORDER NO. AN82-00

DATED JUNE 1980

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be investigated by appropriate technical personnel and action will be taken as required. Receipt of all forms will be acknowledged; however, if you require a detailed reply, check here. ☐

FROM: NAME _____ DATE _____

TITLE _____

COMPANY _____

ADDRESS _____

_____

PLEASE FOLD AND TAPE–
NOTE: U. S. Postal Service will not deliver stapled forms

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 39531 WALTHAM, MA02154

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486

Honeywell

Honeywell

Honeywell Information Systems
In the U.S.A.: 200 Smith Street, MS 486, Waltham, Massachusetts 02154
In Canada: 2025 Sheppard Avenue East, Willowdale, Ontario M2J 1W5
In the U.K.: Great West Road, Brentford, Middlesex TW8 9DH
In Australia: 124 Walker Street, North Sydney, N.S.W. 2060
In Mexico: Avenida Nuevo Leon 250, Mexico 11, D.F.

27662, 1680, Printed in U.S.A.

AN82-00