Virtual Machine CMS/TSO Pipelines
Procedures Macro Language
Modification Level 1.0110

July 18, 2006

John P. Hartmann

IBM Danmark A/S

jphartmann@vnet.ibm.com (from the Internet)
JOHN at EMEAVM1 (from VNET)
Abstract

This book describes techniques and macros used to write Virtual Machine CMS/TSO Pipelines, IBM Program Offering 5785-RAC.

The idea is to implement the abstractions of high-level languages with assembler macros. This increases programmer productivity, avoids some types of errors, and reduces the likelihood of other errors.

Modules contain procedures which in turn contain procedure work areas and procedure bodies whose code is structured after Dijkstra’s control flow constructs: alternatives, iteration, and selection.

Though clearly required, a macro library supporting a structured control flow is not sufficient for a successful migration to the high-level abstraction; the programmer must at all times see the program at the higher level:

- While editing, an XEDIT macro formats the program to show the control flow structure.
- Macros generate code to support the abstraction during assembly.
- A cross-reference of procedures and calls is generated.
- The listing is inspected for valid, but questionable usage.
- The listing is processed to remove detail below the level of the abstraction: for instance, generated instructions and labels.
- A database of modules and programs defines what needs to be re-generated when a module is changed.
- In the event of an ABEND, the running procedure is identified.

Disclaimers

The assembler macros and the listing post-processor described in this book are shipped as part of CMS/TSO Pipelines. However, the Program Offering only supports the macros to assemble an unmodified version of the Program Offering; the listing post-processor is supplied as a sample filter package.

EXECs and XEDIT macros described here are not supplied as part of the Program Offering; nor is the procedure to inspect the program listing.

This book has not been submitted to formal IBM review; the views presented are solely my own.

Acknowledgments

Gerald Davison pointed out that 0/0 is defined in an assembler expression with the value 0. This is used in the branch optimiser to determine if two instructions are adjacent. (X/X is zero when X is zero and one otherwise.)

My most devout student, John C. Lynn, read zillion drafts of this book and made many helpful comments and raised questions that consistently improved it.

Lynghy, January 1997, j.


Reprinted for pdf March 2003.

The specification in this book applies to the macros in the files PIPEIF COPY, PROC COPY, STROPTP COPY, and GENINST COPY in Version 1 Release 1 Modification Level 9 of CMS/TSO Pipelines.

© Copyright International Business Machines Corporation 1989, 2003. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
Contents

Preface ................................................................. vii
Who Is this Book for? ....................................... viii
Contents of Book ........................................ viii
Publications ................................................... viii

Chapter 1. Introduction ........................................... 1
There Is more to It than Macros ......................... 2
Summary ......................................................... 2

Chapter 2. Introducing the Language ......................... 3
The Macros ....................................................... 3
Register Conventions ......................................... 3
Defining a Module ............................................ 4
On Data .......................................................... 4
On Procedures .................................................. 5
    Contents of Procedures .................................. 5
    Declaring a Procedure ................................... 6
    The Procedure Work Area ................................ 8
    Calling a Procedure ...................................... 10
Macros to Control Flow ...................................... 10
    Alternatives .............................................. 10
    Select a Case ............................................ 12
    Iterate ................................................... 13

Chapter 3. A Sample Program .................................... 15
Chapter 4. Creating and Editing a Structured Program .... 16
    Syntax Directed Editing .................................. 16
    Generating Instruction Templates ......................... 16
    Formatting Assembler Source Files ...................... 16

Chapter 5. Assembling a Module ................................. 17
Machine Requirements .......................................... 17
Program Directories ........................................... 17
Macro Libraries ................................................. 18
Assembling a Module with Updates ......................... 18
Inspecting the Listing for Compliance ..................... 18
Listing Post Processor ......................................... 19
    Controlling the Post Processor ........................... 19
Printing the Listing ........................................... 20
Work Area Allocation .......................................... 20

Chapter 6. EXEC and XEDIT Macro Reference ................ 21
EXECs ........................................................... 21
    XI - Edit an Assemble File ............................... 21
    NUFIX - Create a New Update for a Module .......... 22
   XFIX - Edit an Existing Fix in a Module ............. 22
    ASMM - Assemble a Module without Updates ....... 22
    MLU - Assemble a Module with Updates ............. 22
    LINTOK - Store Accepted LINT Errors ............... 22

© Copyright IBM Corp. 1989, 2003
XEDIT Macros .............................................................. 23
PROFILE ......................................................................... 23
CHACT - Define Change Activity ........................................ 23
Q - Insert Structure Template ............................................. 23
SP - Insert a SPACE Instruction ........................................... 23
EJ - Insert an EJECT Instruction ........................................... 24
ASCM - Make a Block Comment .......................................... 24
AS - Add Stars to Lines ....................................................... 24
CONT - Set Continuation Character ...................................... 24
NOCONT - Remove Continuation Character .......................... 24
Formatting the Program ...................................................... 25
FMT - Format Text in Block Comment .................................. 25
REXSTR - Format for Flow Structure ................................... 25
INTSTR - Format a Procedure ............................................. 26
Z - Structural Formatting from Prefix Area ............................ 26
PXWA - Determine Work Area Required for External Procedure ... 26

Chapter 7. Assembler Macro Reference ................................. 27
General ........................................................................... 27
The Structure of a Module ................................................ 27
MODBEG - Open a Module ............................................... 28
MODEND - Close a Module ............................................... 29
Declaring References to Procedures .................................... 29
EXTPROC - Define a Procedure in Another Module ............... 30
WXTPROC - Define a Procedure in Another Module ............... 30
FWDPROC - Define a Procedure later in a Module ............... 31
PDBGCODE - Begin Section of Debug Code ......................... 31
The Structure of a Procedure ............................................ 31
PROC - Open a Procedure Declaration ................................ 32
DC and DS assembler Instructions in the Procedure Work Area .... 34
PBEGIN - Open Procedure Body ........................................ 34
GOTOEXIT - Go to Epilogue Code ..................................... 35
PCALL - Call a Procedure ............................................... 35
PEXIT - End of a Procedure Body ...................................... 36
PROCEND - Declare the End of a Procedure ......................... 37
#PUSEWA - Declare that a Work Area is Available Outside Procedure ... 37
PLTORG - Generate a Literal Pool .................................... 37
Additional Procedure Macros ........................................... 37
Macros to Specify Structured Flow of Control ....................... 38
IF - Conditional Execution .............................................. 38
ELSE - Alternative ......................................................... 38
FI - Close IF ............................................................... 39
COND - Conditionally Execute an Instruction ....................... 39
ASSERT - Ensure a Particular Condition Code ....................... 39
CASBEG - Begin a Selection List ...................................... 39
CASITEM - Declare One Case .......................................... 40
REPEAT - Open a Loop .................................................... 41
LEAVE - Leave a Loop ..................................................... 41
CONTINUE - Iterate a Loop ............................................. 41
UNTIL - Declare the End of a Loop .................................. 42
The Branch Optimiser ..................................................... 42
Storage Management Macros ........................................... 43
FREE - Obtain Storage .................................................... 43
FRET - Release Storage .................................................. 43
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONDSUP</td>
<td>Conditionally Suppress Code for Particular Storage Management</td>
<td>44</td>
</tr>
<tr>
<td>Maintaining Flags</td>
<td></td>
<td>45</td>
</tr>
<tr>
<td>FLAGDEF</td>
<td>Define a List of Flags</td>
<td>45</td>
</tr>
<tr>
<td>FLAGEQU</td>
<td>Define a Set of Flag Bits</td>
<td>45</td>
</tr>
<tr>
<td>FLAGTM</td>
<td>Test a Flag</td>
<td>45</td>
</tr>
<tr>
<td>FLAGON</td>
<td>Turn a Flag On</td>
<td>45</td>
</tr>
<tr>
<td>FLAGOFF</td>
<td>Turn a Flag Off</td>
<td>46</td>
</tr>
<tr>
<td>FLAGXI</td>
<td>Invert Setting of a Flag</td>
<td>46</td>
</tr>
<tr>
<td>FLAGSET</td>
<td>Set all Flags in a Byte</td>
<td>46</td>
</tr>
<tr>
<td>SCALAR</td>
<td>Declare Identifiers in an Enumerated Range</td>
<td>47</td>
</tr>
<tr>
<td>Generalised Instructions</td>
<td></td>
<td>47</td>
</tr>
<tr>
<td>Instructions Referencing Storage</td>
<td></td>
<td>47</td>
</tr>
<tr>
<td>Miscellaneous Generalised Instructions</td>
<td></td>
<td>48</td>
</tr>
<tr>
<td>INCR</td>
<td>Increment</td>
<td>48</td>
</tr>
<tr>
<td>DECR</td>
<td>Decrement</td>
<td>48</td>
</tr>
<tr>
<td>ZIP</td>
<td>Clear Registers</td>
<td>48</td>
</tr>
<tr>
<td>#LAL</td>
<td>Load Address and Length into Register Pair</td>
<td>49</td>
</tr>
<tr>
<td>GENOFS</td>
<td>Generate Offsets to Labels in a DSECT</td>
<td>49</td>
</tr>
<tr>
<td>Chapter 8. Writing CMS/TSO Pipelines Stages in Structured Assembler</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>Procedure Bodies in a CMS/TSO Pipelines Stage</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>The Syntax Exit</td>
<td></td>
<td>52</td>
</tr>
<tr>
<td>Main Procedure</td>
<td></td>
<td>53</td>
</tr>
<tr>
<td>Defining the Stage</td>
<td></td>
<td>54</td>
</tr>
<tr>
<td>PIPDESC</td>
<td>Declare a Pipeline Stage</td>
<td>54</td>
</tr>
<tr>
<td>Codiing an Entry Point Table as Constants</td>
<td>57</td>
<td></td>
</tr>
<tr>
<td>PIPEPT</td>
<td>Define an Entry Point Table</td>
<td>57</td>
</tr>
<tr>
<td>PIPEPTEN</td>
<td>Entry in an Entry Point Table</td>
<td>57</td>
</tr>
<tr>
<td>PIPEPTED</td>
<td>End of an Entry Point Table</td>
<td>58</td>
</tr>
<tr>
<td>Sample CMS/TSO Pipelines Stage</td>
<td></td>
<td>58</td>
</tr>
<tr>
<td>Coding a Syntax Program</td>
<td></td>
<td>60</td>
</tr>
<tr>
<td>The Abstract Machine</td>
<td></td>
<td>60</td>
</tr>
<tr>
<td>Sample Syntax Expressions</td>
<td></td>
<td>61</td>
</tr>
<tr>
<td>Coding a Program for the Syntax Machine</td>
<td></td>
<td>61</td>
</tr>
<tr>
<td>Chapter 9. Assembler Errors and MNOTES</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>Errors Reported by the Assembler</td>
<td></td>
<td>64</td>
</tr>
<tr>
<td>Addressability</td>
<td></td>
<td>64</td>
</tr>
<tr>
<td>Undefined Symbols</td>
<td></td>
<td>64</td>
</tr>
<tr>
<td>Invalid Duplication Factor</td>
<td></td>
<td>64</td>
</tr>
<tr>
<td>MNOTES</td>
<td></td>
<td>65</td>
</tr>
<tr>
<td>Appendix A. Extract from a Program Listing</td>
<td>73</td>
<td></td>
</tr>
<tr>
<td>Appendix B. Miscellaneous Reference Information</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>SYSPARM String</td>
<td></td>
<td>75</td>
</tr>
<tr>
<td>Format of Program Directory Records</td>
<td></td>
<td>75</td>
</tr>
<tr>
<td>The Assembler Directory, ASM PGMDIR</td>
<td></td>
<td>75</td>
</tr>
<tr>
<td>The Object Module Directory, TXT PGMDIR</td>
<td></td>
<td>76</td>
</tr>
<tr>
<td>The Load Module Directory, MOD PGMDIR</td>
<td></td>
<td>76</td>
</tr>
<tr>
<td>Appendix C. Using Material Shipped with CMS/TSO Pipelines</td>
<td>77</td>
<td></td>
</tr>
<tr>
<td>Licensing Issues</td>
<td></td>
<td>77</td>
</tr>
</tbody>
</table>
Preface

This book describes procedures and macros useful to write assembler programs with a high-level-language approach. CMS/TSO Pipelines was implemented using these techniques.

Who Is this Book for?

The book is intended for programmers and service personnel who:

- wish to understand CMS/TSO Pipelines source code,
- modify CMS/TSO Pipelines, or
- write programs using the high-level abstraction.

A working knowledge of CMS as well as IBM System/370 machine instructions and assembler language is assumed.

Contents of Book

[Chapter 1, “Introduction” on page 1] introduces the concepts and explains why they were developed.

[Chapter 2, “Introducing the Language” on page 3] gives an overview of the abstraction and introduces the macros that implement the abstraction.

[Chapter 4, “Creating and Editing a Structured Program” on page 16] explains how to edit programs that use the macros.

[Chapter 5, “Assembling a Module” on page 17] explains how to compile a program.

[Chapter 6, “EXEC and XEDIT Macro Reference” on page 21] defines the syntax and semantics of REXX procedures that are used; these programs are not shipped with CMS/TSO Pipelines.

[Chapter 7, “Assembler Macro Reference” on page 27] is the reference for the macros. This constitutes the language definition for the abstraction.

[Chapter 8, “Writing CMS/TSO Pipelines Stages in Structured Assembler” on page 50] describes how to write a pipeline stage using the macros.

[Chapter 9, “Assembler Errors and MNOTEs” on page 64] lists the MNOTEs issued when errors occur and explains what they mean. MNOTEs are ordered by identifier.

Appendices include a sample page from a listing, background reference information, and instructions on how to extract the macro library and post-processor from the CMS/TSO Pipelines product tape.
Publications

Chapter 1. Introduction

Algol 60 showed the way, but it was not until the end of the nineteen-seventies that Pascal and other modern block structured languages became widespread and were used for systems programming.

There were no CMS implementations generally available at that time; when Pascal/VS was made available it became clear that its run-time environment was not suitable to writing CMS nucleus extensions and other system-type software. And even if it were, CMS’ single address space makes it difficult to devise a way to handle multiple stacks that can grow independently for concurrent programs.

But the abstraction of procedures and control flow is desirable also for writing assembler, the only systems programming language available for CMS. Many errors are prevented within a structural frame normally associated with structured programming. Programming in a structured fashion reduces complexity: the programmer thinks of structural building-blocks instead of individual branches of a spaghetti program. By far the greatest advantage of structured programming is in the maintenance process where it makes the control flow obvious.

CMS/TSO Pipelines macros for the programming infrastructure (as opposed to the programming interface between pipeline filters and the pipeline dispatcher) perform functions in these three areas:

- Declare procedures. With the exception of two dispatcher modules, the CMS/TSO Pipelines consists of programs that contain procedures. A procedure has one entry point and one return instruction. Procedures can be nested as in Pascal.
- Define program flow. The normal structured programming constructs are supported.
- Provide generalised instructions. For example, to load a value into a register independent of the length of the field (one to four bytes).

The macros describing control flow do not depend on an underlying operating system; they can be used in any program assembled with Assembler H Version 2. Changing the macro that generates branches, it should be possible to port these macros to alien architectures: 3705, 3600, 8100, etc.

The macros handling routine entry and exit deal with save areas: they implement the calling conventions using a storage management system. The calling conventions are close to the OS variety (type-1 linkage): the caller provides the save area and the callee saves the general registers. Storage is obtained and released by macros sensitive to the environment the program is being assembled for. CMS and MVS are routinely used. It is also possible to write programs that run on both CMS and MVS and select the appropriate storage management at run time.

The code generated is often better optimised than can safely be done by hand: a peculiarity of the way the assembler evaluates an expression is exploited to avoid many redundant branches, for instance a branch to a branch.

The control flow macros were written in 1979 to 1980 with a review in 1983. The procedure control macros evolved over the period from 1981 to 1984 with minor enhancements in 1988.
There Is more to It than Macros

Just providing a macro library is not enough to support structured programming in assembler; all stages of the program development cycle must support the abstraction:

- Editing the source or update file. A syntax directed editor or a parser for the Live Parsing Editor would be ideal, but there is none such: we shall have to make do with an XEDIT macro to format a program indented according to its structure; other XEDIT macros support the structured constructs, for instance with templates.

- During assembly, macros generate machine code from the high-level macros.

- At the end of the listing, the end-of-module macro writes a cross-reference of procedures and procedure calls in a module, including calls to external procedures.

- A lint procedure inspects the listing for questionable practices. For example, references to storage that do not use a base register. The programmer can accept such flagged statements; an indication to that effect is stored so that the statement is not flagged in subsequent assemblies.

- Before the listing is printed, a listing post-processor removes information that is irrelevant to the structured view of the program. For instance, all generated labels are removed from the cross reference. The object code generated for a macro instruction is shown along with the macro instruction; what assembler instructions are used to generate the code is in general irrelevant. Because it is a post-processor, the programmer can still inspect the listing file from the assembler and has thus access to the complete story when it is needed.

- A run-time environment can include optimised pro/epilogue routines to allocate work areas and discard them automatically, or the code can be generated in-line.

- An ABEND exit traps problems at run time and invokes a REXX/IOS3270 browser to display the PSW and general register set, the current save area, the procedure entry point (to show its name), and the failing instruction. Storage at a given address is also displayed. This browser can also be called from a program or command level.

Summary

Few write assembler by choice. But if you have to, this is the way to do it.
Chapter 2. Introducing the Language

The language constructs have a distinct Pascal flavour, though of necessity the syntax must conform to assembler.

A module contains one or more procedures. A procedure has three sections: a work area which looks like a DSECT, a procedure body with the executable code, and a static area for read-only constants.

The flow of control within the procedure body is described by macros supporting the usual structured concepts of alternatives, selection, and iteration.

A procedure can be nested in a containing one; work area allocation is in general more efficient when calling a nested procedure than when calling an outer one.

The Macros

The macros described in this book were used to write CMS/TSO Pipelines and are shipped as part of it. There are three types of macros; they are shipped as separate copy files:

PROC Module and procedure definition macros, including macros to define a procedure’s work area; macros to pass control between procedures.

STROPTP Control flow macros define how control flows through a procedure. Alternation, selection, and iteration are defined.

GENINST Generalised instructions generate an operation that is appropriate to the size of the object being operated on. This avoids, for instance, storing a fullword into a halfword field. Macros define scalars and flags, test and manipulate flags without the programmer needing to know where the flag is defined, avoiding errors where a flag is set or tested in the wrong byte.

Register Conventions

The general registers shown below are reserved for use by the procedure macros; other general registers are not used, or are preserved across functions.

<table>
<thead>
<tr>
<th>Figure 1. Register Conventions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>15</td>
</tr>
</tbody>
</table>

OS linkage conventions are used: the caller allocates the save area; the callee saves and restores registers unless instructed not to do so. Module base is not established, each entry point establishes addressability locally; this allows a module to grow beyond 4K without a painful search for an additional base register. Forward procedure calls are generated with
a BAL instruction unless FAR=YES is specified; external calls and backwards calls generate a L and a BALR.

There is one departure from OS linkage conventions: The first word of the save area for a level-1 procedure is reserved to contain the size of the block of storage in doublewords.

---

**Defining a Module**

A module contains module declarations and procedure declarations.

A module begins with MODBEG which defines the supported types of storage management (if any). An eye-catcher with the date and time of assembly and a copyright notice are generated.

Module declarations include data areas and external procedures. In some cases it is necessary to declare that a procedure in the module is called by forward procedure calls; these declarations are in the module declarations as well.

Procedure declarations declare the executable code in the module. There can be any number of procedures; procedures can be nested to optimise the allocation of work areas.

The last instruction in a module is MODEND. It generates prologues, epilogues, a procedure cross-reference, and the assembler END instruction.

---

**On Data**

Data fields can be declared in these contexts:

- In a data area DSECT. Such data areas typically refer to control blocks that are passed by pointer reference. Instances of such a control block can be declared in a procedure work area or in a static area. Data areas are normally declared before the first procedure.

- In a procedure work area. A particular instance of the field is created whenever the procedure is called; it is released when the procedure returns. The procedure work area is declared between the PROC and PBEGIN macros. The variable is addressable in the procedure where it is declared; it may or may not be addressable in nested procedures; it can be made addressable in other places.

- In the static data area in a procedure, between the PEXIT and PROCEND macros. One instance of the variable is allocated and initialised at assembly time for all instances of the procedure. The program is not re-entrant if a variable in the static data area is modified. Variables in the static area of a procedure are addressable from containing procedures and (within the 4K limit) from level-1 procedures (and their nested procedures) declared earlier in the module.

- Statically in the module before, between, or after procedures. Use PDATA to allocate an area of static storage after all procedures declared so far. Such an area is addressable from procedures before it. Use PLTORG before PDATA to allocate the literal pool before a large data area at the end of the module.

The notion of writable static in the C language is not supported; do not declare variables outside DSECTS or procedures (or use PDATA). On the other hand, variables in the caller’s work area can be referenced by a nested procedure; allocate a global data area in the level-1 procedure and pass a pointer when going outside the module.
On Procedures

There are these types of procedures:

- A level-1 procedure (a main procedure). It allocates a save area.
- A level-1 procedure that allocates its work area at an address passed by the caller. This is similar to dynamic storage areas used by compilers.
- A nested procedure. It allocates its save area and work area (if any) in the containing procedure’s work area.
- A procedure that saves registers, but does not allocate a save area.
- A procedure that neither saves registers, nor allocates a save area or a work area.

The general procedure saves general registers at entry and restores them at exit. It allocates a save area and a work area dynamically at procedure entry; it releases the storage at procedure exit; it can call itself recursively.

Optimisation is possible in special cases, some of which can be determined without programmer assistance; other optimisations require the programmer to declare that the program is in some way restricted: Clearly, there is no need to allocate a save area for a procedure that does not call further and uses no work area.

To implement a small routine that can be called efficiently, the programmer may specify that general registers need not be saved on entry (SAVE=NO) and that the procedure must not allocate a save area (SAREA=NO). The caller needs no save area to call such a procedure.

Contents of Procedures

The work area

This is a list of DS and DC instructions. It can contain macros that expand parameter lists with such instructions, for instance FSCB. The work area is allocated dynamically at run time usually with a call to the operating system storage management. DCs in the work area are initialised by the prologue for the entry point. (The DC and DS instructions have been OPSYNed in the work area; the work area is a DSECT as far as the assembler is concerned.)

The work area can be in two parts: one that is directly addressable (up to 4K), and one part (normally up to a total of 32K) that is not directly addressable. The second part can be used for buffers larger than 4K.

The work area is in two parts because the first part of nested procedures’ work areas must be addressable from the calling procedure’s work area. The maximum aggregate work area size depends on the calling structure: it must be possible to address an inner work area from an outer work area. Which depends on how registers are declared as base registers for work areas, and the flow through the procedures leading to the one in question.

The Procedure Body

The procedure body contains executable instructions. It begins with PBEGIN and ends with PEXIT. A procedure on level 1 (the outermost) can have multiple procedure bodies sharing a common work area and a common set of nested procedures.
A prologue is generated before the procedure body to save general registers, initialise the
work area, and establish addressability; the epilogue generates return linkage and restores
general registers. The programmer’s code is between the two.

**Static Data Area.**
Though it may contain any instruction, it normally has read-only constants, and
instructions that are the target of execute (EX).

**Nested Procedures**
Nested procedures, if any, are declared between PEXIT and PROCEND for the containing
procedure.

**Declaring a Procedure**
A procedure is declared between PROC and PROCEND macros. A procedure declaration
inside another procedure is said to be nested within a containing procedure. Nested proce-
dures can only be called from the path of containing procedures. The work area of a
containing procedure can be made available to the nested one. Internal procedure calls
must conform to a static call hierarchy which is enforced by the macros. A procedure on
any level may call a level 1 procedure which then allocates a new work area. The call
path for internal procedures must be descending through the tree describing the structure.
It is permissible to call from level n directly to level n+m.

The save area and work area for a nested procedure are allocated in the containing
procedure’s work area. The work area for the outermost procedure can be pre-allocated by
the caller or obtained and released dynamically.

PBEGIN declares the beginning of the procedure body; it is implied when no work area is
defined. PEXIT declares the end of the procedure body and the beginning of the static
section; it can be elided if the procedure has no nested procedures and no static area.

Procedures are re-entrant unless the programmer writes code that modifies the procedure
body or the static data area.

When a procedure is declared it is not in general known if the procedures it calls require a
save area or not. In most cases, the expansion of prologues and epilogues are deferred to
the end of the module. Location counters are defined by PBEGIN, but left empty for these
two sections; the procedure body coded by the programmer goes into a location counter
between the two, thus ensuring that everything does get into the right place eventually.

GOTOEXIT is used to terminate execution of a procedure before the PEXIT instruction. This
is often used as an error exit.

A procedure name can be declared without the procedure parts in these ways:
- Use EXTPROC when the procedure is in some other module.
- Use WXTPROC when the procedure is in some other module and a weak external refer-
  ence is to be generated.
- Use FWDPROC to declare a procedure using SAREA=STACK or SAREA=HLLSTACK, to
  which there are forward calls.

The example in [Figure 2 on page 7](#) shows a sample procedure hierarchy. Four procedures
are declared, A..D, in the hierarchy shown in [Figure 3 on page 7](#).
A PROC WORKBASE=R10
W1 DS CL8
PBEGIN, Begin of code for level 1 procedure
....
PCALL B Call nested procedure
PEXIT, Return level 1

B PROC DCL=NO No local work area
....
PCALL A Valid recursive call
PEXIT
PROCEND, End of procedure B

C PROC
W2 DS CL80 Local data
PBEGIN
...
PCALL D Call level 1 procedure D
PROCEND, End of procedure C... PEXIT assumed
PROCEND, End of procedure A

D PROC DCL=NO
...
PROCEND RC=0

Figure 2. Sample Procedure Hierarchy

A and D are not nested. The procedures can call each other as shown in the table below:

<table>
<thead>
<tr>
<th>From</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>B</td>
<td>OK</td>
<td>n/a</td>
<td>n/a</td>
<td>OK</td>
</tr>
<tr>
<td>C</td>
<td>OK</td>
<td>n/a</td>
<td>n/a</td>
<td>OK</td>
</tr>
<tr>
<td>D</td>
<td>OK</td>
<td>n/a</td>
<td>n/a</td>
<td>OK</td>
</tr>
</tbody>
</table>

The PCALL macro has adopted the convention that if

'A' || left(ep,min(7,length(ep)))
is defined, it is assumed to be a fullword containing the address of the routine. This is a handy way to support a branch vector if one is available and generate normal external addresses when there is no branch vector.

**The Procedure Work Area**

A procedure’s work area has a standard 18-word save area at the beginning; the save area is followed by the data structure declared between **PROC** and **PBEGIN**, if any. Each procedure with a save area has a DSECT for the save area, and the work area extension, if present.

Variables declared with **DC** instructions in the work area are initialised from constants generated in the static data area when a procedure is called. This puts a few restrictions on what is initialised in the work area:

- Constants are generated in the static section, so address constants cannot refer to addresses in the procedure work area; use **LA** and **ST** to initialise pointers to data in the work area.
- The aggregate length of adjacent constants must be 256 or less. The assembler reports a length error in the prologue when a run of constants is longer than 256 bytes. Insert **DS 0C** to break a long run of constants.
- Length fields and repetitions may be written as expressions in parentheses if the expression can be evaluated by the assembler *at the time the statement is met*. You cannot reference symbols resolved later in the program; if you do, the assembler reports errors when **MODEND** computes work area sizes. Try to rearrange the way the work area is laid out or move the declaration of DSECT data areas to the beginning of the program between **MODBEG** and the first procedure.

The save area allocations for the procedure A in [Figure 2 on page 7](#) is shown in Figure 4.

![Figure 4. Work Area Allocation for Sample Hierarchy](image-url)
Allocating the Work Area

A level-1 procedure can allocate and release the work area by calls to the operating system storage manager in the prologue and epilogue. This may be too costly; the programmer can allocate the work area in other ways:

SAREA=STACK makes PCALL load the address of an area of storage to use in general register 1; SAREA=HLLSTACK causes PCALL to place the address of the storage area to use in the forward pointer of the save area (at offset 8). Storage required by the called procedure is allocated if the body of the called procedure is declared later in the module; a FWDPROC macro is required in this case. The caller must perform a similar action when other linkage mechanisms are used, for instance if the address of the routine is in a table and the routine is called with BASR.

If no other information is available, a save area (72 bytes) is allocated for the called procedure. The storage required is known if the procedure is declared later in the module. For an external procedure, specify the work area requirement (in bytes) as the third sub-operand of the EXTPROC or WXTPROC statement declaring the external procedure.

An escape mechanism exists to let the programmer specify macros to allocate and deallocate the work area (see SAREA=). CMS/TSO Pipelines uses

SAREA=(,#PRABOVE,#PRFRET)

to call the pipeline storage manager to allocate storage. This makes the prologue independent of the host system.

The work area for a nested procedure is normally allocated in the caller’s work area. This is simple enough when there is only one caller. When a nested procedure is called from several containing procedures, the work area is allocated relative to the work area of the caller closest to the level-1 procedure or the nearest outer procedure that has declared a register for its work area. The level-1 procedure is normally declared with WORKBASE= to ensure that the nested procedure’s work area can be allocated.

Addressing the Work Area

You do not normally have to do anything to make a work area addressable; USINGs are issued as appropriate in prologues and before static data areas to expose work areas that are known to be addressable. These USINGs are dropped in the epilogue and after the end of the static data area.

A procedure’s own work area is addressable from general register 13.

A containing procedure’s work area may be available to a nested procedure. Without programmer assistance, the caller’s work area is available in a nested procedure if the nested procedure does not allocate a save area and has only one caller.

The programmer designates a general register to hold the base of a procedure work area with the WORKBASE= operand on PROC. Which of the work areas are available to a nested procedure is determined by static analysis of the call tree; a nested procedure can declare that a register no longer contains the address of a work area (HIDE= on PROC) or it can use the same register for its own work area. In both cases, the containing procedure’s work area is not made available.

Though all labels in work areas are defined in the whole module (IBM System/370 assembler has no facilities for name scoping), the availability of a work area in a nested proce-
dure is controlled by USING instructions; an invalid reference leads to an addressability diagnostic.

**Calling a Procedure**

The macro PCALL is used to call a procedure. It generates appropriate linkage instructions for calls to declared procedures. For instance, when calling a procedure declared with SAREA=STACK, it loads general register 1 with the address after its own work area.

A call to a (so far) undeclared procedure is assumed to be to a forward call to a procedure later in the module. The procedure can be nested or on level-1; it is assumed to save the general registers.

**Macros to Control Flow**

Control flow is declared with macros that reflect the usual model of alternatives, selection, and iteration. For each construct there is a beginning macro and an ending macro, so that the specification of an inner block is automatic.

The structured programming constructs can be nested to any depth, though it is not recommended to have large deeply nested structures. The macros ensure that the structure is well formed (for instance that a FI ends an IF).

REXX control flow instructions have these equivalents:

- **If**
  - IF, ELSE, and FI. There is no THEN macro.

- **Select**
  - CASBEG and its associated macros implement selection and branch vectors.

- **Do**
  - REPEAT and UNTIL define iterative loops. Since the end of an IF is explicit, there is no need for the Do group.

- **Leave**
  - LEAVE.

- **Iterate**
  - CONTINUE.

- **Return**
  - Implied by PEXIT. Use GOTOEXIT to return before the end of the procedure (for instance an error return).

**Alternatives**

Figure 5 shows the building block for decision making. The condition code is tested. If the specified condition is met, then the statements between IF and ELSE are executed. If the condition does not hold, then the statements between ELSE and FI are executed. ELSE is optional.

Note that IF does not develop the condition code, it tests the current setting of the condition code. Figure 5 shows a program that increments general register 1 if general register 0 is zero; it clears general register 1 if the condition code is not zero.

```
<table>
<thead>
<tr>
<th>Macro</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LTR</td>
<td>R0,R0</td>
<td>Test register 0</td>
</tr>
<tr>
<td>IF</td>
<td>ZERO</td>
<td>It is zero:</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>Increment and set CC</td>
</tr>
<tr>
<td>ELSE</td>
<td></td>
<td>When it isn't zero</td>
</tr>
<tr>
<td>SR</td>
<td>R1,R1</td>
<td>Clear</td>
</tr>
<tr>
<td>FI</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

*Figure 5. Sample If*
The IF statement generates a conditional branch to the list of statements after the ELSE statement. The ELSE statement (if present) generates an unconditional branch to the FI statement followed by a label for the conditional branch generated by the IF statement. The actual code generation is more complex to cope with nesting of structures and to optimise branches.

Figure 6 shows a short-hand for conditional execution of a single instruction. The instruction can be any machine instruction or a macro with up to four positional operands.

<table>
<thead>
<tr>
<th>LTR</th>
<th>R0,R0</th>
<th>Is there something?</th>
</tr>
</thead>
<tbody>
<tr>
<td>COND</td>
<td>ZERO,A,R1,=F'1'</td>
<td>No, count</td>
</tr>
</tbody>
</table>

Figure 6. Sample Conditional

Programming note: To reduce nesting, and/or type expressions can be written in a way that separates the test from the action, as these examples show:

```
CLI X,0
COND NOTEQUAL,CLI,X,20
COND NOTEQUAL,CLI,X,50
IF EQUAL
  ... ... The statements to do
  ... ... if x is 0, 20, or 50
FI
```

Figure 7. OR-type Expression

The construct in Figure 7 generates the condition code for the expression x=0 | x=20 | x=50 and then tests it. The code generated by the optimiser for the two branches implied by the COND statements branch directly into the IF body.

And-type logic is implemented in a similar way:

```
CLI X,0
IF EQUAL
  CLI Y,0
  IF EQUAL
    CLI Z,0
  FI
FI
IF
  ... ... The statements to do
  ... ... if x, y, and z are 0
FI
```

Figure 8. AND-type Expression

In the sample in Figure 8, branches are generated past the last FI statement. How much further depends on containing structures.

The last IF block could have been inserted after the compare to Z; the code generated would be the same. However, separating the two things minimises the effect of a change to the way the condition is computed. This makes for small update files to apply changes:
In the example above a further test can simply be inserted after the CLI for Z. Had the resultant action been immediately after the test for Z, then the scope of the change would have included the action and the indent for the last if block would also change, and thus the whole construct would have to be replaced.

**Select a Case**

CASBEG declares the beginning of a selection; CASEND is the end of the list of cases. Each case is preceded by CASITEM.

There are two types of expansion, sequential and vectored. In the sequential case, tests are performed in the order of the CASITEM macros. In the vectored case, a value in a general register is used as an index into a branch table which is filled in by each CASITEM macro; wild branches cannot occur.

When testing sequentially, an instruction is generated with an operation code and the first operand common to all tests; the second operand is specific to each test. This is well suited to testing flags declared with FLAGDEF or a small subset of the possible values of a scalar.

```assembly
READORWRITE PROC ,
   FLAGDEF R,W,C
   PBEGIN ,
   ... compute the flag value
   CASBEG FLAGTM,,ONE
   CASITEM R Reading?
       FSREAD FSCB=FSCB
   CASITEM W Writing?
       FWRITE FSCB=FSCB
   CASITEM , Default
       L R15,=-8'
   CASEND ,
   PROCEND RC=(R15)
```

*Figure 9. Case*

The CASBEG macro generates no code for the sequential test case; it stores the arguments in global variables, which are referenced in subsequent CASITEM macros. In this case the instruction used is FALGTM, which only requires one operand, so the second positional operand is omitted.

Any number of CASITEM macros may be coded. The expansion generates suitable tests and branches to select one of the clauses, or none. CASITEM with no operands indicates the default case. It must be the last one; no further CASITEMs can be coded after one without operands.

The vectored mode is suited to decode a scalar declared with SCALAR. It generates a branch vector for all potential items; the vector points to the default case initially. Addresses in the vector are filled in as the individual cases are declared. The item selected depends on a value in a general register. The case values are specified much like Pascal scalars, with the assembler performs static bounds check at compile time.
Looping is done with a REPEAT..UNTIL clause. Iteration is controlled by keywords which are close to the System/370 looping instructions used to perform the iteration: BCT, BXLE, BXH, and ICM. It is also possible to terminate the loop at a specified condition code.

Figures 10, 11, and 12 illustrate various loop constructs:

**Figure 10. Vectored Case**

**Figure 11. Repeat (Do Until)**

Figure 11 shows one way to scan a string for leading blanks. The IF ensures that the string is not null; the loop compares characters until the string is exhausted, or a non-blank character is met. The base is incremented and the count decremented at the end of the loop.

The REPEAT statement generates the label for the loop, and if requested a conditional branch and a BXH instruction for loop control.

Do forever is written as UNTIL without operands.

A condition can be coded on REPEAT. This is shorthand for enclosing the loop in an IF for this condition. (Figure 11 can be simplified in the same way.)

**Figure 12. Repeat (Do While)**
The macros LEAVE and CONTINUE are used to terminate and iterate a loop, respectively. A specific outer level of loop may be left or iterated if an identifier is specified.

The construct called a 1.5 iteration loop (where the exit condition is tested in the middle) is coded as shown in Figure 13.

```
REPEAT ,
  LTR R3,R3 More to do?
  LEAVE COND=NOTPOSITIVE
  PCALL DOCHUNK
UNTIL , Forever
```

*Figure 13. 1.5 Iteration Loop*

CONTINUE is used to transfer control to the UNTIL macro; this can reduce nesting in a complex loop.
Chapter 3. A Sample Program

Figure 14 shows the sample Hello, World! program written for VM.

```plaintext
HELLO  TITLE ' Hello, World! ' *
         COPYRIGHT IBM Danmark A/S' *
***************************************************************
* *
* Write a greeting to the console of the virtual machine. *
* *
* Change activity: *
* 31 Aug 1994 +++ Make XA capable. *
* 17 Oct 1989 New module. *
*
***************************************************************
SPACE 2
HELLO  MODBEG FREETYPE=NONE No storage required
GO    PROC ENTRY=NO,SAREA=NO
LINEWRT DATA=HELLO
PEXIT  RC=(R15)
HELLO  DC C'Hello, World!'
PROCEND ,
MODEND GO
```

Figure 14. Hello, World!

The positional operand on MODEND is the name of the procedure to invoke. Without it CMS would execute the eye-catcher generated by MODBEG.
Chapter 4. Creating and Editing a Structured Program

Syntax Directed Editing

A syntax directed editor understands the structure of the language and ensures that the program being edited is always syntactically correct.\(^1\)

For example, if the programmer wishes to insert an IF statement, a key is hit to tell the editor this. A template appears for an if statement with then and else clauses. The reserved words are protected so the programmer cannot change the structure of the program, and since they are not typed, typing errors cannot occur.

This approach applies directly only to programs being created. The internal representation of the program is in terms of the syntax tree, not the statements of “normal” programs. This approach involves a conversion if used to change existing programs.

There is some debate whether a syntax directed environment is useful for real programs and programmers; be that as it may, XEDIT does not lend itself to an easy implementation of such a function. Comparatively crude macros are used to insert templates; to compensate there is a sophisticated XEDIT macro to format the code.

Generating Instruction Templates

XEDIT prefix macros insert templates for procedures and the control flow constructs, as well as SPACE and EJECT assembler instructions.

Formatting Assembler Source Files

Using macros to designate the structure of programs means that labels only appear on the PROC statement beginning a procedure and in the work area and the static data area; in the procedure body the outermost level of indent can be in column 2. Convention is to indent by two columns for each nesting level of control flow structure.

Indention is done by an XEDIT macro that understands enough of assembler syntax to translate what the assembler wants in uppercase and leave character strings and comments in mixed case. This is not as trivial as one might think; first of all, SETB expressions may contain blanks around relational operators. Character constants and things spanning lines are not that simple either:

```
DC (L'ABC)CL(L'XYZ)' character string '
```

The selection level of the statement is set to the indent level (set select). Thus, the set display XEDIT sub-command may be used to suppress innermost constructs (or outermost ones for that).

\(^1\) Tim Teitelbaum and Thomas Reps, The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. CACM 24.9, 563, September 1981. This article describes a total programming environment including interpreter.
Chapter 5. Assembling a Module

A module is normally maintained with multi-level update. When assembling, updates are applied to the base module producing a temporary source file to be processed by the assembler to make an object module. A post-processed listing is written to the virtual printer if no errors are detected during assembly. The listing file is inspected to find suspicious constructs (this is often called LINT).

The update log is put in front of the object module which is renamed to have a file type according to the highest update level applied. The object module is put in a TXTLIB when the update level is the default.

Object modules for different environments can be generated from a single source module. When no updates are applied (or the default text is requested), the object module is put in an object library which is specific to the environment. When multilevel updates are used, separate control files, with unique text levels, must be used to assemble for the different environments.

Machine Requirements

To use the procedures described here, you must have a temporary disk as your A-disk; 10 cylinders 3380 (or equivalent) is recommended. Permanent files must be stored on some other disk. Virtual storage in excess of 2 megabytes is required; 4M is recommended.

Program Directories

Files with file type PGMDIR are used to store information about modules in your system. Information is added with NEWMOD XEDIT.

Each module (ASSEMBLE file) can be assembled for several types of storage management and test options. Three file names are used to define a simple database which can be spanned over several minidisks. The complete definition of the contents of program directory files is in "Format of Program Directory Records" on page 75; here is an overview.

ASM

This file has a line for each combination of module, storage management, and test options. These module and storage management are put in column 2 onwards as a single token with periods between the components.

The next token is the disposition of test code. The name of the TXTLIB to receive the object module follows; a period means that it should be left with file type TEXT if no updates were found.

The remainder of the record lists the macro libraries to use during assembly in addition to the macros defined in this book.

TXT

For an object module, list the load modules that should be regenerated when the object module changes. The first token is the TXTLIB name and the module name with a period between them; the rest of the line lists the names of the load modules to regenerate.

MOD

Lists for each load module which object modules are to be included. The first token is the load module name, the next is the type of module, and the rest of the line lists the object modules in the order they must appear. It is possible to include specification from a separate file and to issue CMS commands as part
of this process, for instance to generate an object module for an entry point table.

Macro Libraries

Macros are stored in packed files with file type COPY. Updates are stored in individual update files. When needed for assembly, the file must be unpacked and any relevant updates applied. The macro library is stored on the temporary A disk and discarded when the disk is detached or the virtual machine is logged off.

Assembling a Module with Updates

MLU EXEC uses information in the program directory, and a control file (file type CNTRL) to:

- Build required macro libraries. This may also include applying updates based on the control file specified for the module.
- Unpack the ASSEMBLE source file and apply updates as indicated by the control file.
- Assemble the module and create an object module. The object module is stored in a TXTLIB if it would get the default file type. Object modules for other update levels are kept as separate files with file type TXTxxxx.
- Process the listing to inspect the generated code for compliance with a defined subset of instructions (e.g., no privileged operations) and to check for instructions with a zero base register. Deviations are stored in a file with file type LINTERRS, except accepted deviations which are read from a file with file type LINTOK. Deviations not present in the new module are deleted from the reference of acceptable deviations.
- Process the listing file through the post-processor.

Inspecting the Listing for Compliance

The listing file is processed to extract instructions that are of a dubious nature:

- Privileged instructions.
- SVC instructions.
- Instructions referencing storage with base register zero. This may be a reference to a field in NUCON, but it is often caused by a SS instruction with SI operands.
- xCM instructions with a mask of 7. This might have trouble in 31-bit addressing mode.
- xxCL instructions. The length is modulo 16M, also in 31-bit mode.
- LTR of general register 12 into itself. This is a test to determine the addressing mode.
- Shift values are within the bounds for the type of shift. (At most 31 bits for single; 63 bits for double.)
- MH with a fullword-aligned operand that is not a Y-type constant.
- Procedures for which the work area does not include the work area for a procedure with SAREA=STACK. (The callee is declared before the caller.)
- LA into general register 0 with a displacement of 7, 15, 23, or 31. This catches many potential modulo 16M problems.
Listing Post Processor

The post-processor makes the listing useful in the context of structured programming. The first part of the listing is the program listing; the ESD directory is deferred to the directories section.

The result is a listing that is in the context of the macros used while retaining the information required for debugging at the machine level if that is required.

1. Blank pages are deleted.
2. The left-hand part of a line for a macro call in open code gets the generated code of a selected line from the macro expansion.
   Highest priority is an executable instruction, lowest is a location counter display.
3. Generated statements are suppressed.
   Many macros only expand in one line of executable code, so here we lose no information.
4. Comment statements with a continuation is displayed on one line starting at the left margin. It was felt that it would be fun to experiment with this; it requires an XEDIT macro to generate suitably contorted comments. The line is really too long for comfortable reading; so this should be exploited only with XEDIT macro that can format this contorted format in multicolumn!

The directories and cross references are processed to minimise the paper needed to print them. Directories are printed with two spaces between them in this order:
5. The procedure cross reference showing which procedures are declared, how work areas are addressable, and who calls who. This information is obtained from MNOTE statements generated by MODEND.
6. The External Symbol Directory is printed three entries to a line sorted alphabetically.
7. The relocation directory is compressed with six entries to a line.
8. The standard cross reference is printed with lines run together with a few spaces between entries. Names are highlighted by overprinting. Generated labels, register references (R0 through R15), and fullword numeric constants are deleted from the cross reference.
9. The diagnostic directory is reduced to one line containing the record statistics when no statements are flagged.

The minimum size of a listing is two pages, one with the code, and one with the directories.

Controlling the Post Processor

A comment beginning \*%PRINT controls the post-processor so that generated code can be shown in parts of the listing, when desired. The second word on the line must be in column 9 after a single blank; it defines what to do:

- **ALL**: Copy input to output without inspection.
- **SINGLE**: Process macro expansions as described above (the default).
- **PUSH**: Save the current setting of ALL/SINGLE on a ten-level stack.
- **POP**: Restore the previous setting from the stack.
Printing the Listing

The listing post-processor produces a listing file with 1403 carriage control in the virtual printer. Names in directories are emphasised by overprinting. The SPOOL file is in hold status.

Managing these SPOOL files requires a bit of thought. Depending on the programmer’s temperament, several listings may be created for a module as it is being debugged; obsolete listing files should be discarded before printing.

Printing on an impact printer is simple enough: the file is changed to NOHOLD which releases it to CP SPOOL. For other devices, a post-post-processor may be needed.

On a 3800-1 or a 3800-3 in compatibility mode, overstruck words should be printed in a different font, so the listing file needs a character arrangement table code in column 2. The built-in program `overstr` generates a descriptor line with ‘00’x in column 1 in front of lines with overstruck words. `optcdj` separates the contents of the data record after each descriptor record so that normal words are written with the default character arrangement table and overstruck characters are written with a different character arrangement table.

Work Area Allocation

The work area map allows the programmer to verify that the work area allocation for a module is reasonable. Here is an example from a module that contains three level-1 procedures:

```
1  PIPBLCBK A8 :wi . . . . .
2  VALIDATE_LENGTH 48 : w . . . . .
1  PIPBLCDB 90 :wi . . . . .
2  PIPASALR 48 : w . . . . .
2  PIPASANR 48 : w . . . . .
1  ADDRDW 2A0 :wwwwwi . . . .
2  VALIDATE_LENGTH 48 : w . . . .
```

A summary of the individual module’s work area requirements is kept in a database, whence it can be determined automatically how much work area to allocate when calling an external procedure that uses SAREA=STACK or SAREA=HLLSTACK.
Chapter 6. EXEC and XEDIT Macro Reference

This chapter describes execs and XEDIT macros that are useful when editing a program. The most important one (and the largest by far) is REXSTR which formats the structure with indents. There are several scaffolding macros around it, and other macros to format text. These other macros are presented as a starter set; more could be done in this area.

Best results are obtained if you ensure that you have “CASE Mixed Ignore” when editing ASSEMBLE and COPY files.

EXECs

These execs are handy when editing modules. The first one is used to create a module and maintain source when the module is edited directly; other execs are used to manage change by multi-level update.

XI - Edit an Assemble File

XI <fname>

If a file of type ASSEMBLE exists and there are no error messages stored, the file is edited without interference.

When files with types ASSEMBLE and ERRORS exist for the module, the latter is assumed to be the SYSTERM output form a previous assembly. The two files are edited with split screen and the source file is positioned at the first line in error.

When the module does not exist, a dummy module is created. The alteration count is set to 0.

ZZ TITLE ' &*ust.itle *
COPYRIGHT IBM Danmark A/S'
COPY PGMID

ZZ TITLE ' &*ust.itle *
COPYRIGHT IBM Danmark A/S'
COPY PGMID

Figure 15. XI ZZ

The cursor is put at the title.
NUFIX - Create a New Update for a Module

Generate a new fix for a module.

```
NUFIX <module>[.<baseftype>] <ftype> <description>
  [% [%<controlfile> [<auxfile>]]]
```

A line with the file type and description is added to the specified auxiliary control file for the module (AUXFIX is the default). A CHACT is stacked to enter the description in the change activity log. A dummy update file is created; XFIX is called to edit the fix.

The base file type needs only be coded when it is not ASSEMBLE or COPY.

XFIX - Edit an Existing Fix in a Module

Edit an update file against a module.

```
XFIX <module>[.<baseftype>] [ftype]
  [[<xedit options>] [%<controlfile>]]
```

The base file type needs only be coded when it is not ASSEMBLE or COPY.

ASMM - Assemble a Module without Updates

```
ASMM <modules> [[<options>]]

<modules> ::= <module> | <module> <modules>
<module> ::= <file name>[.<storage management>[.<testoption>]]
<options> ::= <option> | <option> <options>
<option> ::= NOLINK | NOPRINT | NOTXTLIB
```

MLU - Assemble a Module with Updates

```
MLU <module> [<controlfile> [<program directory>] [[<options>]]

<options> ::= <option> | <option> <options>
<option> ::= NOERASE | NOPRINT | DISK | NOLINK
```

LINTOK - Store Accepted LINT Errors

```
LINTOK <module>
```
The errors in the file with type LINTERRS are processed and merged with the contents of the file with type LINTOK on disk H.

XEDIT Macros

A number of XEDIT macros simplify the creation and change of structured code.

PROFILE

The profile should issue these commands when it determines that the base file type is ASSEMBLE or COPY:

- set case mi
- set serial all 10000
- set trunc 71

CHACT - Define Change Activity

CHACT <comments>

A line is added to the change activity summary in the first block comment in the module. If no change activity summary exists, one is created at the end of the first block comment.

* Change activity:
  * 21 Nov 1987 A sample change

Figure 16. CHACT A sample change

Q - Insert Structure Template

Use this macro to insert a template after the line it is entered for. Specify as an operand the type of statement you wish:

- p A procedure is desired. The cursor is placed for you to enter any operands. Then backtab to the beginning of the line and enter the procedure name.
- i An IF-ELSE-FI construct is desired. The cursor is placed ready for you to fill in the condition.
- c Case construct is desired.
- m Additional CASITEM line is desired.
- r Repeat/Until template desired.

The macro may be issued both from the command line and from a prefix area.

SP - Insert a SPACE Instruction

SPACE [<number>]

A SPACE <number> assembler instruction is inserted. The default is 1.
The macro may be issued both from the command line and from a prefix area.

**EJ - Insert an EJECT Instruction**

An EJECT assembler instruction is inserted.

The macro may be issued both from the command line and from a prefix area.

**ASCM - Make a Block Comment**

Frame a number of lines (one by default) as a block comment.

```
ASCM [<target>]
```

Unless already there an EJECT is inserted in front of the block comment and a SPACE 2 is added after it.

The target must be a single blank-delimited token. The default is to format a single line.

The macro may be issued both from the command line and from a prefix area.

**AS - Add Stars to Lines**

Add stars to the sides of lines, for instance lines added to a block comment.

```
AS [<target>]
```

The macro may be issued both from the command line and from a prefix area.

**CONT - Set Continuation Character**

An asterisk is put in column 72 to indicate a continuation.

```
CONT [<target>]
```

The target must be a single blank-delimited token. The default is one.

The macro may be issued both from the command line and from a prefix area.

**NOCONT - Remove Continuation Character**

A blank is put in column 72.

```
NOCONT [<number>]
```

The number of lines can be specified; it is one by default.
Formatting the Program

Several macros are available to make the program pretty and convert instructions to uppercase, leaving comments and strings in mixed-case.

**FMT - Format Text in Block Comment**

Formats the text in a block comment so the text appears in columns 3 through 62. This leaves room for the S&D code in columns 64 to 71.

\[ \text{FMT} \quad [<\text{count}> \quad [<\text{left margin}> \quad [<\text{right margin}>]]] \]

“FMT” understands about other file types than ASSEMBLE and COPY.

The macro may be issued both from the command line and from a prefix area.

**REXSTR - Format for Flow Structure**

This XEDIT macro reads lines from the file and formats them to respect the control flow structure and the assembler requirements for names, and so on, in uppercase. Complete statements are formatted; most macro formats are supported.

\[ \text{REXSTR} \quad [<\text{target}> \quad [<\text{level}>]] \]

When formatted, a statement is only replaced if it is different from the original one. This is important when creating update files; the aim is to “touch” as few lines as possible to make a minimal update file.

This macro takes 0, 1, or 2 arguments.

The complete module is re-formatted when no arguments are specified.

The first token specifies a target. This may be of any form as long as it has no imbedded blanks. With this variety, lines are formatted from the current line up to but not including the target.

If specified, the second token must be numeric. It specifies the starting indent level. The outermost level is 1. Level 4 has the operation code in column 10. Between MACRO and MEND statements the default starting column is 10; outside it is 2. However, if the formatting starts inside a macro then you must ensure the correct level is set up.

If an operand ends with a trailing comma, you are prompted to determine if a continuation was indeed desired or if other typing errors had occurred.

Lines are assigned a selection level corresponding to their indent level. To leave out anything on e.g., level 3 and further, you should enter the command “set display 0 2”. Beware that this interacts with the selection level set by the ALL command.
INTSTR - Format a Procedure

This is an interface macro to REXSTR. It formats from the most recent previous PROC statement through the nearest PEXIT or PROCEND whichever occurs first.

No arguments are required.

Z - Structural Formatting from Prefix Area

- Z without operands invokes INTSTR to format the procedure the line is in.
- With numeric operands (one or two) it invokes REXSTR with these operands to format a number of lines. With two operands the second one is the indent level to start at.
- When the last (or only) operand is a special character, lines are formatted to the indent level of the line before the one where the prefix command is issued. With a number as the first operand, that number of lines are formatted. Without a number, one statement is formatted.

PXWA - Determine Work Area Required for External Procedure

Position the current line on an EXTRPOC macro that has Stack or Hllstack as the second sub-operand. PXWA will verify the work area size specified against the database and increase it if necessary. PXWA also works in the PIPEVECT and PIPEVEC2 members of PIPEIF COPY.
Chapter 7. Assembler Macro Reference

General

This chapter describes the syntax and semantics of the macros.

The macros are kept simple to make the program easy to read. The programmer is assumed to have a good knowledge of the at least the general instruction section of the System/370 Principles of Operation manual.

All generated labels have a prefix of ###. The prefix ###$ is reserved for the programmer or macro writer to declare labels that are to be removed from the cross reference.

The Structure of a Module

MODBEG
[
<procedure reference declarations>]
[
<data areas>]
[
<procedure list>]
MODEND

Figure 17. Structure of a module

The macro MODBEG is used to declare the type of storage management to be used. Register equates are defined.

Literal pool, procedure prologues, and procedure epilogues are generated by the MODEND macro. No END statement is required.

Imported data areas and local data areas are most conveniently declared in the data areas section so that the attributes of a datum are available to the generalised instructions. Use normal COPY, DSECT, DS, and USING instructions.
MODBEG - Open a Module

Code MODBEG as the first statement of your program, after TITLE and comments.

```
<name> MODBEG FREETYPE=<freespec>
  [AMODE={24|31|ANY}]
  [RMODE={24|31|ANY}]
  [CNOTE={YES|NO}]
  [TRANSIENT={NO|(YES,<procedure>)}]
  [TARGET={370|XA}]
  [PRINT={NOGEN|GEN}]
  [SELFRELOCATE={NO|YES}]
  [NOSTORAGE={ABEND|RETURN}]
```

FREETYPE Lists the types of storage management supported by the module. Several options may be coded in parentheses; in this case the actual storage management is specified by SYSPARM.

- **NONE** The module has no FREE or FRET macro instructions. Procedures needing save areas are declared with SAREA=STACK or SAREA=HLLSTACK; or they specify macros to allocate and deallocate the work area in the second and third suboperands of the SAREA operand.
- **CMS** CMS user storage is desired. DMSFREE is used; the program must reside below the 16M line.
- **CMSSYS** CMS nucleus storage is desired. DMSFREE is used; the program must reside below the 16M line.
- **CP** Calls to DMKFREE and DMKFRET are generated.
- **OS** GETMAIN and FREEMAIN are used.
- **DYNAM** The CVT is inspected to determine if the invocation is under CMS or MVS/XA. When running on CMS, CMS storage is obtained. When running in MVS/XA storage is obtained below the 16M line.
- **AMODE** The addressing mode required for the module. ANY is the default except for CMS where the default is 24.
- **RMODE** The residence mode required for the module. ANY is the default except for CMS where the default is 24.
- **CNOTE** Code NO. The default, YES, generates a copyright notice that makes IBM Denmark the copyright owner.
- **TRANSIENT** Use the default, NO, unless the module is entered at the beginning of the CSECT instead of at one of its entry points. Code YES and the name of the procedure to receive control if the module is entered at the beginning as done by, for instance, CMS transient routines; this generates a branch around the eye-catcher (see also MODEND).
- **TARGET** Code the target architecture. The default is 370. When the target is 370, OPSYNs are set up to map BAS and BASR into BAL and BALR. This allows you to write programs which are easier to port from /370 to XA.
- **PRINT** Controls the printing options for the code generated by MODBEG. NOGEN is the default.
SELFRELOCATE Code YES if you wish PCALL to generate self-relocating code.

NOSTORAGE Specify the action to take if storage cannot be allocated for a procedure work area. Code RETURN to cause the prologue to return with code -9 when it is unable to allocate storage. The default is to request the operating system to terminate the task when there is insufficient storage to allocate the procedure work area.

The name of the module is remembered; a START instruction is generated. The module name, date, time of assembly, and (unless suppressed) a copyright notice are generated as constants.

Code XIENT= for modules that are entered at the beginning, for instance modules running in the CMS transient area. This generates a branch around the eye-catcher.

MODEND - Close a Module

This macro should be the last line of the program. It generates prologues and epilogues that were not previously generated, and a procedure call cross reference.

Though assembled at this late stage, prologues and epilogues are put in location counters so they appear correctly around the procedure body code. This allows the pro/epilogues to be generated knowing if the procedure calls further.

```
MODEND [<entry point>]
[GENEND={YES|NO}]
[PRINT={NOGEN|OFF|GEN|DATA}]
[XREF={YES|NO}]
```

<entry> Code the name of the entry point to put on the END card if an entry should be defined, for instance the main entry point for a CMS command.

GENEND Code NO if you provide the END card. YES is the default.

PRINT Control PRINT setting while generating prologues and epilogues. Code GEN to have these printed. The default is NOGEN.

XREF Code NO to suppress the procedure cross reference. The default is YES.

Specify an entry point as the positional parameter in cases where you would have done so on an END statement.

The pro/epilogues are normally assembled with “PRINT NOGEN”; to print the expansion code PRINT=GEN or PRINT=DATA.

Declaring References to Procedures

References between procedures are normally implied by the PCALL macros. External procedures must be declared unless AUTOEXT= is coded on the PCALL instruction. All weak external references must be declared explicitly.

Forward references (a call to a procedure declared later in the module) must be declared when the callee is a procedure declared SAREA=STACK or SAREA=HLLSTACK; the work area required by the callee is allocated in the caller’s work area when the callee procedure is declared after the calling procedure.
**EXTPROC - Define a Procedure in Another Module**

External level 1 procedures are declared with this macro. An EXTRN instruction is generated for each entry point.

```
EXTPROC <proc-item>,...
```

```
<proc-item> ::= <name> | (<name>,<type>[,<wa-size>])
<type> ::= YES | NO | STACK | HLLSTACK
```

- `<name>` The external name of the entry point to be defined.
- **YES** The procedure requires a 18-word save area addressed from general register 13 to store the general registers.
- **NO** The procedure does not save general registers.
  It is the programmer's responsibility to ensure that external procedures defined as `<name>,NO` do not require a save area on entry. To ensure that no save area is generated, `SAVE=NO` and `SAREA=NO` should be coded on the declaration of such procedures.
- **STACK** The procedure requires a 18-word save area addressed from general register 13 to store the general registers. General register 1 points to an area of storage that can be used for work area. PCALL loads the address of the doubleword following the caller's work area into general register 1 when calling a procedure declared `SAREA=STACK`.
- **HLLSTACK** The procedure requires a 18-word save area addressed from general register 13 to store the general registers. The forward pointer in the save area allocated by the caller points to an area of storage that can be used for work area. PCALL puts the address of the doubleword following the caller’s work area into the forward pointer of the save area when calling a procedure declared `SAREA=HLLSTACK`.

If it is specified, the third suboperand is the size (in bytes) of the work area required by the procedure. The default is one save area (72 fullwords).

**WXTPROC - Define a Procedure in Another Module**

External level 1 procedures are declared with this macro. An WXTRN instruction is generated for each entry point.

```
WXTPROC <proc-item>,...
```

```
<proc-item> ::= <name> | (<name>,<type>[,<wa-size>])
<type> ::= YES | NO | STACK | HLLSTACK
```

The operand list is defined with EXTPROC above.
FWDPROC - Define a Procedure later in a Module

This macro is used to specify that a procedure is declared later in the module. It is only needed when the STACK or HLLSTACK option is desired; forward references to other types of procedures do not require explicit declaration. This macro generates no code.

FWDPROC <proc-item>,...

proc-item ::= <name> | (<name>,<type>[,<wa-size>])
type ::= STACK | HLLSTACK

The operand list is defined with EXTPROC above.

PDBGCODE - Begin Section of Debug Code

Input lines are ignored unless the SYSPARM includes TEST GEN to indicate that code should be generated.

PDBGCODE [<number>]

If a number is coded, that number of input cards are suppressed. If no argument is coded, lines are suppressed to the next one beginning “*.end” in lowercase. Lines are included in the listing file without assembly when TEST LIST is in the SYSPARM.

The Structure of a Procedure

The macros described below generate prologues, epilogues, and inter-procedural calling sequences.

Save areas are allocated dynamically; procedure work areas are allocated and initialised in an appendage to the save area.

To optimise work area allocation, the programmer can define a hierarchy of nested procedures where the save area and work area for each nested procedure are allocated in the containing procedure’s workarea. This allows for efficient subroutine linkage with the restriction that a nested procedure can only be called from procedures in its hierarchical path.

<label> PROC
[<label>] DC <dc operands>
[<label>] DS <ds operands>
PBEGIN
... code and procedure calls ...
PEXIT
<static constants>
<procedures>...
PROCEND [<label>]

Figure 18. Structure of a procedure

A procedure has several parts shown in Figure 18:
1. The procedure header defined by the macro PROC.

2. The procedure work area is declared between the PROC macro and the PBEGIN macros. It may be elided.

   The area is allocated on doubleword boundary. DC and DS assembler instructions can be coded in any order in this section, but initialisation is optimised if DCs are adjacent. DCs are initialised with a MVC; a dummy DS must be inserted to break up a specification that is longer than 256 bytes.

   Do not code machine instructions in the work area; they are not initialised. Either code the instructions as hexadecimal and S-type constants, or move a prototype piece of code to the work area.

3. The procedure body code; it ends with the PEXIT macro instruction.

4. Any local constants or executed instructions may be declared after the PEXIT statement. Initialisation values for DC statements are allocated first in this pool.

   Note: This area is addressable from containing procedures, but not from nested procedures even if they are declared before the static instructions. It follows that any static data that need to be addressable from all routines in a call hierarchy must be declared in the last procedure on the innermost level.

   Warning: Special care should be taken when instructions instructions in the static area are issued with execute (EX). Such instructions are assembled with the USINGs in effect in the static section; the base register is loaded correctly only in the procedure in which the static area is part, but not in containing ones. Nevertheless, the instruction can be addressable from containing procedures. There seems to be no way to make the assembler detect such invalid usage.

5. Any number of imbedded procedures.

   Placing the internal procedures here, instead of at the beginning as one would expect from Algol, is done to allow the work area definition to be complete before internal work areas are defined. It also allows forward calls to generate a BAS instruction.

6. The end of a procedure is declared with a PROCEND macro. Omit PEXIT if there are no internal procedures and no static data.

**PROC - Open a Procedure Declaration**

This macro designates the beginning of a procedure. The label must be coded; it specifies the name of the procedure.

```
<label> PROC [ENTRY={YES|NO}] [SAVE={YES|NO}] [DCL=NO] [WORKBASE=<reg>] [HIDE=<reg>|(<reglist>)] [SAREA=<sareaspec>] [FWDONLY={NO|YES}]
<sareaspec> ::= NO | STACK | HLLSTACK | ([<size>],<opcode>,[<opcode>])
```

**ENTRY** This operand is only inspected on level-1 procedure declarations. Code NO to declare that the procedure can only be called from within the module. The default is YES; this generates an ENTRY instruction for the procedure label.
SAVE Code NO if the contents of registers 0 through 11 are not to be saved on entry and are to be passed back to the caller upon return. Code YES to have general registers saved on entry and restored on exit.

DCL Code NO if no work area is to be declared; do not code an explicit PBEGIN.

WORKBASE Code the number of a general register (2 through 11) used to address the work area. The work area is made addressable in nested procedures. The default is to use general register 13 as the base for the work area in which case it is not available to nested procedures that allocate a save area.

HIDE Code a list of general registers declared as WORKBASE by containing procedures when you modify these registers in a procedure.

SAREA Code the save area requirement for the procedure. YES, the default, means that a save area is allocated if required. NO means that no save area must be allocated and that only procedures declared SAVE=NO may be called (this is enforced). STACK and HLLSTACK are only valid for a level-1 procedure. SAREA=STACK means that the caller provides the address of a pre-allocated save area in general register 1. SAREA=HLLSTACK means that the caller provides the address of a pre-allocated save area in the forward pointer of the save area. PCALL sets these pointers up automatically.

FWDONLY Code YES to declare that no calls are made to this procedure in the remainder of the program.

The ENTRY operand is used only on level 1 procedures. Default is that such procedures are known globally.

The WORKBASE operand must be specified if the procedure work area is to be made available to nested procedures. Nested procedures inherit workbases set up in containing procedures if it is guaranteed from analysis of the call flow that these containing procedures will be visited on all calls to the nested procedure in question.

HIDE is specified on an nested procedure when a register or a list of registers should not be inherited. The list enumerates the registers; they cannot be specified as a range. It is not an error to HIDE a register that has not been declared as a WORKBASE in a surrounding procedure.

DCL=NO is specified to indicate that a procedure work area is not declared. A PBEGIN macro must not be coded if this is the case. A save area is allocated unless DCL=NO (or SAREA=NO) is coded; coding a null work area always causes a save area to be allocated.

SAVE=NO may be coded to indicate that the procedure wishes all changes to general registers 0 through 11 fed back to the calling procedure and that general registers 14 and 15 are not used. Special optimisation takes place if the procedure does not call further.

Note: It is not an error to code SAVE=NO and still include a PCALL in the procedure, but in this case general registers 12, 13, and 14 are saved.

SAREA=NO declares that the procedure does not call routines that save the general registers, and thus needs no save area. This allows the procedure to inherit the caller’s work area in general register 13, if all calls are from this one procedure. It is an error to code a PCALL in a procedure declared as SAREA=NO and SAVE=NO. Do not code a PBEGIN macro after a PROC macro with SAREA=NO.

You can allocate the save area by other means; you can code three operands:
SAREA=(<length>,<opcode1>,<opcode2>)

This declares that <opcode1>, when executed without operands, loads general register 1 with the base of an area that is at least <length> bytes. The number of double words required is provided in general register 0. If coded, it is ensured that <length> is sufficient for the work area; it may be elided. If coded, <opcode2> is generated to release the area on exit from the procedure; no action is taken to release the work area if <opcode2> is not coded.

SAREA=STACK specifies that the calling routine provides the address of an area to use in general register 1 when calling this routine. SAREA=HLLSTACK indicates that the called routine expects the address of a work area to be stored in the forward pointer in the save area (offset 8). PCALL stores the address of the next available save area into the save area before calling a procedure that is declared to use SAREA=HLLSTACKForward calls to such a procedure work only when the callee has been declared by FWDPROC.

The procedure work area follows the PROC macro unless DCL=NO or SAREA=NO is specified.

**DC and DS assembler Instructions in the Procedure Work Area**

DC and DS are OPSYNed to macros between PROC and PBEGIN. The areas declared with DC are initialised at each procedure entry, unless disabled by INITWA=NO on PBEGIN. The contents of areas defined with DS are not initialised.

Relocatable address constants referring to the current location counter (*) should be avoided in DC instructions: the star refers to the address of the constant in the constant pool, not to the variable declared. Constants referring to addresses in the work area must be initialised explicitly with LA and #ST.

DC and DS macros are coded exactly like DC and DS assembler instructions, but the aggregate operand length must be 256 or less because it is passing through a SETC symbol. (Separate DC instructions cannot be generated for each positional operand since that would defeat fractional lengths.) Because macros are invoked, the operands may be coded in the alternate operand format which is not supported by the corresponding assembler instructions.

**PBEGIN - Open Procedure Body**

This macro designates the end of the procedure work area and the beginning of the code section.

```plaintext
PBEGIN [EP=<name>]
[ENTRY=NO]
[INITWA=NO]]
```

**EP**  
Code the entry point name on the PBEGIN macro when a level-1 procedure has multiple procedure bodies.

**ENTRY**  
Code NO to suppress an ENTRY instruction to be generated for the entry point when EP= is coded.

**INITWA**  
Code NO to suppress the prologue initialising constants in the work area.

A level 1 procedure may have several entry points (PBEGIN/PEXIT pairs) sharing a common work area and internal procedures. In this case the entry point name is coded following
the ENTRY= keyword. The default is to generate an ENTRY statement, but that is suppressed by coding ENTRY=NO. These entry points allocate identical work areas and have a common hierarchy of internal procedures.

When used with SAREA=STACK on the PROC macro, INITWA=NO indicates that the entry point is not to initialise the DCs in the procedure work area; this allows data values stored by a previous invocation of this or another procedure body to be saved between calls (assuming that the caller does not re-use the area or release it).

Coding EP= forces the prologue and epilogue to be generated in-line. This may be necessary to avoid addressability problems in large modules when the macros allocating the procedure work area reference literals.

**GOTOEXIT - Go to Epilogue Code**

This macro is coded to (un)conditionally return from the procedure. It generates a branch to the PEXIT macro expansion unless the procedure is declared SAVE=NO and SAREA=NO: then a BCR is generated to return immediately.

```
GOTOEXIT [<return code>]
[COND=<if-condition>]
```

If a positional operand is specified, the value is loaded into general register 15 with a LA instruction. The return code is always loaded into general register 15, also if it turns out that a return is not to be made.

Code COND= to specify that the exit is only performed if the specified condition prevails; the default is an unconditional return.

**PCALL - Call a Procedure**

```
PCALL <ep>
[PARM=<parameter list address>]
[AUTOEXT={(YES|YES,<type>)}]
[FAR=YES]
[GEN=NO]
```

Suitable linkage is generated to the specified procedure. AUTOEXT= causes an EXTPROC <ep> or EXTPROC (<ep>,<type>) to be generated if the name has not been defined so far.

If neither FAR=YES nor AUTOEXT is specified and the name is not yet defined, the call is assumed to be to a procedure declared later in the module, and a BAS is generated. Otherwise a load and a BASR are generated.

If left('A'<ep>,8) exists it is assumed to be the label of a fullword containing the address of the entry point.

The parameter address is loaded into general register 1 with a LA if PARM is coded.

GEN=NO declares that no code should be generated because the entry point is called by other means, e.g., via a branch table.
**Note:** General register 1 can be used to transmit a parameter unless the callee has declared SAREA=STACK; in this case, general register 1 is loaded with the address of the procedure work area to be used by the callee.

**PEXIT - End of a Procedure Body**

PEXIT designates the end of the executable part of the procedure.

```
PEXIT [RC=<rcspec>]  
  [REGS=<regspec>] 
  [CC=<ccspec>]
```

- `<rcspec>` ::= `<self-defining-term> | (<register>)`
- `<regspec>` ::= `<simple-reg> | (<reg-list>)`
- `<reg-list>` ::= `<simple-reg> | <simple-reg>,<reg-list>`
- `<simple-reg>` ::= `<register> | (<register>,<register>) | (<register>)`
- `<ccspec>` ::= RETAIN | RESTORE | (<register>)

**RC** Defines the return code. The default is 0. A LA instruction is used unless the operand is coded in a parentheses to indicate that the contents of a general register is to be used. Code RC= without operand to restore general register 15 if it was saved.

**REGS** Define general registers to be returned to the calling procedure. Note that SAVE=NO on the PROC macro implies REGS=(R0,R11); REGS= should not be coded in this case. The operand can be a single register, a pair of registers in parentheses, or a list of register specifications.

**CC** Specifies how the condition code is to be set. The default is to test the return code. Code RC= and CC=RETAIN to pass back the current condition code. Code a register number in parentheses to have the value in this register set as the condition code and program mask. CC must not be coded if the procedure has allocated a work area and releasing the work area changes the condition code, or if the program runs in XA-mode and a BALR or BASR is issued when releasing the work area.

The default return code is zero. RC= must be written without an operand if no return code should be fed back.

The condition code to be returned can only be specified if RC is coded without operand.

The general registers returned are specified by the REGS specification. Pairs of registers in parentheses indicate a range. By convention, REGS=(Rx,Ry) means the range from x to y. Registers must be in the range 0..11, and Rx<=Ry. Code REGS=((Rx),Ry) to return two individual registers.
PROCEND - Declare the End of a Procedure

If a positional operand is specified, it is compared against the name of the corresponding PROC macro and a diagnostic is issued if they are not equal.

\[
\text{PROCEND } [<\text{name}>] \\
[\text{RC=<rcspec>}] \\
[\text{REGS=<regspec>}] \\
[\text{CC=<ccspec}>]
\]

The PROCEND macro accepts the same keyword operands as does the PEXIT macro. If no PEXIT macro has been coded for the procedure, then one is generated automatically using the keyword operands coded. The keywords are ignored if the procedure body has already been closed with a PEXIT macro.

#PUSEWA - Declare that a Work Area is Available Outside Procedure

Use this macro when a level-1 procedure is called exclusively from another level-1 procedure and its nested procedures.

\[
\text{#PUSEWA } <\text{procedure}>[,<\text{register}>]
\]

The general register specified is normally coded with WORKBASE= on the PROC statement for the procedure specified, but this is not verified; it is possible to use a different register.

PLTORG - Generate a Literal Pool

Code PLTORG where you would normally have coded a LTORG. The macro ensures that the literal pool is generated after all procedures declared so far, and is thus addressable. Code PLTORG only if you have trouble with addressability; the assembler generates a literal pool at the end of the module.

Additional Procedure Macros

Some of the internal macros may be of use for systems programming type macros.

- Use PDSECT to re-open the work area definition, for instance to allocate a parameter list. It is too late to have constants implicitly initialised by DC at this time, so only DS instructions should be coded in the work area.
  
  In a level-1 procedure you can code the operand LOCTR=1 to open the second part of the procedure work area. This part can be any size.

- Use PCODE to resume the code section after you have used PDSECT.

- Use PDATA to ensure that the following instructions are assembled after all procedures declared so far.
Macros to Specify Structured Flow of Control

Unlike PL/I, the macros imply a block, so there are no requirement for a grouping Do statement.

Note that the following description of the macros ignores the branch optimiser; refer to “The Branch Optimiser” on page 42.

ID= Keyword
All of the flow control macros take a keyword parameter, ID=, to specify an identifier for the macro. This usage is optional.

The macros LEAVE and CONTINUE process this identifier to designate which loop should be iterated or left, while the remaining macros support it to verify that the structure is the intended one.

You may code ID= on the opening macro of a structured construct and not code it on the remaining macros in that construct, but if coded on a macro that does not open a construct then the opening macro must have the same identifier coded. Identifiers need not be unique.

The ID keyword is not described further.

The IF Condition
The condition code is specified in several macros. Only these keywords are recognised (some are more than eight characters): BORROW BUSY CARRY CHANNELBUSY COMPLETE CSWSTORED EQUAL HIGH INCOMPLETE LOW MINUS MIXED NEGATIVE ONE ONES OVERFLOW PAGEINVALID PLUS POSITIVE SEGMENTINVALID UNATTACHED ZERO.

IF - Conditional Execution
The macros IF, ELSE, and FI are used to implement an if-then-else structure.

```
IF <if-condition>
<if-condition> ::= [NOT]<built-in> | (<cond-list>)
<cond-list> ::= <built-in> | <built-in>,<cond-list>
```

The instructions following IF are executed only if the specified condition holds.

ELSE - Alternative
ELSE declares the end of the list of statements executed when the IF condition holds, and the beginning of the list of statements to execute when the condition does not hold. The statement is optional.

```
ELSE ,
```

No positional operands are allowed.
FI - Close IF

This macro declares the end of the scope of a previous IF macro.

```
FI ,
```

No positional operands are allowed.

COND - Conditionally Execute an Instruction

The COND macro specifies conditional execution of a single statement.

```
COND <if-condition>,<opcode>[,<operands>]
<operands> ::= <operand>[,<operand>,[<operand>],[<operand>]]
```

Instructions having more than one operand are coded with up to for positional operands after the operation code, or specified by enclosing the operands in quotes or parentheses. The enclosing delimiter is removed in this case. The single operand “(x)” may be generated by '(x)' or ((x)). This passes through a SETC symbol, so the list appears flat if the operation code is a macro. Keyword operands cannot be coded; you must use IF/FI to specify conditional execution of a macro instruction with keyword operands.

The statement generates this code:

```
IF <condition>
   <opcode> <operand>
FI
```

ASSERT - Ensure a Particular Condition Code

Use ASSERT to test the condition code and force a program check when the condition code is not what it should be. After the test for the condition, a halfword of zeros forces a program check for operation exception. The second operand is stored in the following halfword.

```
ASSERT [<if-condition>],<hex>
```

The assert failure is unconditional if the first positional operand is ommitted (that is, the operand field begins with a comma). This is useful in the default CASITEM clause.

CASBEG - Begin a Selection List

The case statement provides for the selection of one of a list of blocks to execute. There are two types, a repetitive test and a vectored branch via a branch table.

The type of case is defined in the CASBEG macro. The CASITEM macro starts out the definition of each item and the CASEND macro identifies the end of the case construct.
Repetitive test
This variety generates, for each item, a test followed by a conditional branch. The test is often CLI or TM, but any instruction or macro may be coded. Operand 1 may be enclosed in parentheses, so it is really just the left hand part of the operand field for the generated test instruction; the right-hand part is specified in the selector.

The MACRO processor has a few quirks in this area that hit you if you try to generate a macro instruction for the test: SETC makes an operand sublist flat, so when the test is generated, everything is a positional second operand irrespective of the number of commas and equal signs in the operand. Machine instructions appear not to be a problem. A similar remark applies to the condition specification: it cannot be a list of built-in values.

Vectored CASE
The expansion is (1) tests for the value being within bounds, (2) a load of the target from a vector, (3) a branch to the item, and (4) a vector of the offsets to the items, all initialised to point to the default/end.

When an item is defined, the entry in the vector is overlaid with the offset to the generated label. Assembler diagnostics are issued (invalid repetition count) if the item is out of bounds, hence you are insured that a wild branch cannot occur if the program assembles without error.

The general register specified by REG= must be loaded with the encoded value before the macro is executed.

Minimum and maximum values are coded as self defining terms. The default minimum value is 1. A minimum value of literal 0 or 1 reduces the amount of code generated slightly. The SCALAR macro generates a suitable data description if desired.

The operand TEST=NO is intended for decoding where a bit field has already been masked and shifted. The base (MIN=) must be zero; the designated general register must contain four times the index, that is, the offset to the entry in the table for the case. It is assumed that it is known (for instance by masking) that the general register cannot have a value outside the valid range. This option reduces the amount of code generated to a L and a B.

CASITEM - Declare One Case
A CASITEM macro declares a case. With no operands it declares the default case; this must be coded last. The positional operand(s) represent test(s) to carry out, or numbers in the vectored case.
CASITEM [\langle itemlist \rangle]
\langle itemlist \rangle ::= \langle item \rangle \mid \langle item \rangle,\langle itemlist \rangle

In the repetitive test mode, execution proceeds by evaluating the instruction:

\langle opcode \rangle \langle operand1 \rangle,\langle item \rangle

If the condition code is NOT\langle condition \rangle, then the next item is tested.

When the condition is \langle condition \rangle or there are no operands on the CASITEM macro, then
the code for the item is executed. At the end of any item, control passes to the instruction
following the CASEND macro.

Multiple operands on the CASITEM macro is interpreted to mean that the item should be
selected if any of the generated instructions result in the specified condition code. Evalua-
tion is left to right and stops as soon as the desired condition is met.

**REPEAT - Open a Loop**

```
REPEAT \[COND=\langle if-condition \rangle\]
\[BXH=(\langle r1 \rangle,\langle r2 \rangle)\]
```

The COND operand of REPEAT is shorthand for enclosing the REPEAT/UNTIL block in an IF.
Thus, a loop while construct is implemented by this subterfuge:

(evaluate expression)
REPEAT COND=<cond>
...
(evaluate expression)
UNTIL NOT<cond>

**LEAVE - Leave a Loop**

```
LEAVE \[COND=\langle if-condition \rangle\]
\[ID=\]
```

LEAVE exits from the innermost or a containing loop.

When an identifier is coded, the open loops are inspected from the innermost one outwards
until one with a matching identifier is found. Execution continues after the end of this
loop.
CONTINUE - Iterate a Loop

CONTINUE transfers control to an implied label before the UNTIL of this or a containing loop.

When an identifier is coded, the open loops are inspected from the innermost one outwards until one with a matching identifier is found. Execution continues at the UNTIL of this loop.

UNTIL - Declare the End of a Loop

If a positional operand is coded it is taken to be the condition under which control should transfer out of the loop; this test is performed before the other forms of loop control are executed.

BCT takes priority over BXLE.

CHAIN generates an ICM <reg>,15,<address> and code to leave the loop if the new contents of the general register is zero. thus:

UNTIL CHAIN=(R5,NEXTSLOT)

A diagnostic MNOTE is issued if both CHAIN= and a condition are specified, unless I=KNOW is also specified.

Order of evaluation: The operands of UNTIL are evaluated in this order:

- The condition code specified as a positional operand.
- CHAIN= is performed; the loop is left if the new pointer is zero.
- The register in INCR= is incremented.
- BCT= or BXLE= is performed.

The Branch Optimiser

The optimiser tries to avoid a branch to another branch instruction. A peephole optimisation strategy is adopted, and only one active branch target is supported.

The optimiser also optimises an IF immediately following an UNTIL if the conditions are compatible.
The CONTINUE macro of an infinite loop is optimised to the beginning of the loop, as is a conditional LEAVE if the loop is conditional and the conditions specified are suitable.

---

**Storage Management Macros**

The macros FREE and FRET are used to allocate and release storage by calls to the storage manager selected for the assembly. This allows a single source file to be compiled for a number of environments.

**Note:** Programs running as stages under control of CMS Pipelines should use pipeline services to allocate storage and maintain buffers, preferably declaring the stage with the FPEP macro.

**FREE - Obtain Storage**

This macro is used to allocate a piece of storage. The address of the area is returned in general register 1. The number of doublewords allocated is returned in general register 0. This number should be stored by the program to support storage management schemes with a granularity larger than a doubleword.

```
FREE [SIZE=<doublewords>|BYTES=<bytes>] [CLEAR={NO|YES|LONG}] [SETLENGTH={NO|YES}] [BASE=<register>] [MSG=NO]
```

The number of doublewords requested is specified by the SIZE keyword; it may be specified in bytes instead using the BYTES keyword. A LA instruction is used to load the length, so you can only obtain 32K-8 bytes using these keywords. If neither is specified then the number of doublewords must be preloaded into general register 0.

Specify SETLENGTH if you wish the number of doublewords obtained to be stored in the first word of the area.

CLEAR may be specified if SIZE or BYTES have been specified. The area is cleared to zeros. Note that the first word is cleared unless you specify SETLENGTH=YES. CLEAR=YES should be coded for areas shorter than 257 bytes. CLEAR=LONG must be coded to clear a larger area; general registers 0, 1, 14, and 15 are used in the process, so BASE= must be coded.

The address of the newly allocated area is always returned in general register 1; a LR instruction is generated to copy it into the specified general register when you code the BASE keyword.

MSG=NO suppresses CMS messages when there is no more storage available.

**FRET - Release Storage**

This macro is used to return a piece of storage to the free storage manager.

```
FRET [SIZE=<doublewords>|AUTOSIZE=YES] [LOC=<address>|ALOC=<address>]
```

Chapter 7. Assembler Macro Reference
The size of the area to release is obtained in one of three ways:

1. The number of doublewords is obtained from the first word of the area if AUTOSIZE=YES is coded. The length should have been stored as a result of coding SETLENGTH=YES when the area was obtained. This is the recommended approach.

2. A number of doublewords may be specified using the SIZE= keyword.

3. If neither is specified then the number of doublewords must be preloaded in general register 0.

Likewise, the location of the area to release may be specified in one of three ways:

1. Coding the LOC keyword, normally in the form LOC=(<register>), to specify the address.

2. Coding ALOC= generates a load from the specified address.

3. Coding neither means that the address is already in general register 1.

**CONDSUP - Conditionally Suppress Code for Particular Storage Management**

Lines are suppressed depending on the storage management selected for the assembly.

```
CONDSUP <number>,<conditions>[,NOT]

<conditions> ::= <condition> | (<condlist>)
<condlist> ::= <condition> | <condition>,<condlist>
```

The first operand is the number of input records that are to be suppressed. Note that the count is not the number of assembler statements which will be smaller if there are continuations.

The second operand lists the storage management(s) for which the code is to be suppressed.

The action is reversed when the third operand is the keyword NOT. This may be counter-intuitive.

```
CONDSUP 1,CMS,NOT
DMSFREE DWDS=(0) Allocate if CMS
CONDSUP 3,CMS
SLL R0,3 In bytes
GETMAIN R,LV=(0)
SRL R0,3 In doublewords
```

**Note:** CONDSUP does not generate dual-path code to select the appropriate path at execution time.
Maintaining Flags

There are macros to declare and manage flags (bits). A flag is on when the corresponding bit is 1 and off when the bit is 0. The programmer manipulates a flag without specifying the location of the flag, thus avoiding the class of errors where a flag is tested or set in the wrong location.

**FLAGDEF - Define a List of Flags**

```
[<label>] FLAGDEF <names>
   [INIT=<value>]

<names> ::= <name> | <name>,<names>
```

List the names of the flags you wish to define. Up to eight flags can be defined.

INIT Code a self-defining value to initialise the flags; the default is to set all flags off.

**FLAGEQU - Define a Set of Flag Bits**

Several flag bits can be referred to with one name.

```
FLAGEQU <name>,(<flags>)

<flags> ::= <flag> | <flag>,<flags>
```

**FLAGTM - Test a Flag**

```
FLAGTM <name>
   [BASE=<register>]
```

The positional operand is either a single value of a list of values in parentheses.

When designating an explicit base register for the test, the corresponding FLAGDEF macro must have a label and the offset must have been defined by GENOFS.

The flags are tested on by TM.

**FLAGON - Turn a Flag On**

```
FLAGON <name>
   [BASE=<register>]
```

The positional operand is either a single value of a list of values in parentheses.
When designating an explicit base register for the test, the corresponding FLAGDEF macro must have a label and the offset must have been defined by GENOFS.

The flags are turned on by OI.

**FLAGOFF - Turn a Flag Off**

```
FLAGOFF <name>
    [BASE=<register>]
```

The positional operand is either a single value of a list of values in parentheses.

When designating an explicit base register for the test, the corresponding FLAGDEF macro must have a label and the offset must have been defined by GENOFS.

The flags are turned off by NI.

**FLAGXI - Invert Setting of a Flag**

```
FLAGXI <name>
    [BASE=<register>]
```

The positional operand is either a single value of a list of values in parentheses.

When designating an explicit base register for the test, the corresponding FLAGDEF macro must have a label and the offset must have been defined by GENOFS.

The flags are flipped by XI.

**FLAGSET - Set all Flags in a Byte**

```
FLAGSET <name>
    [BASE=<register>]
```

The positional operand is either a single value of a list of values in parentheses.

When designating an explicit base register for the test, the corresponding FLAGDEF macro must have a label and the offset must have been defined by GENOFS.

The flags listed are turned on by MVI; this turns off all other flags in the flag byte.
Declaring a Range of Scalars

**SCALAR - Declare Identifiers in an Enumerated Range**

```plaintext
<label> SCALAR <idlist>
  [FIRST=<selfdef>]
  [INIT=<a-expr>]
  [DECODE=<csect>]

:idlist::= <identifier> | <identifier>,<idlist>
```

When a label is coded, constant of length one is declared with the value coded by the INIT parameter which defaults to zero. No storage is allocated when there is no label on this instruction.

The positional operands are the names of identifiers to be declared. The first one is equated to the FIRST operand which is one by default, incrementing by one for each operand.

If coded, the DECODE operand defines the name of a CSECT where the characteristics of the scalar are defined. Thus, a generalised routine can fill in a message if the scalars have been named in a sensible manner. The format of this control section is:

- 0 A fullword containing the smallest scalar value. (The FIRST operand.)
- 4 A fullword with the largest scalar value.
- 8 A list of doubleword with the name of each scalar.

### Generalised Instructions

Macros are provided to generate the instruction appropriate to the length of the operand, so the programmer needs not be concerned with the details of the declaration of a binary numeric field.

A general register is indicated when the second operand is in parentheses.

To let the macro processor determine the length of an operand, it must be declared prior to use in a generalised instruction.

### Instructions Referencing Storage

- **#A** Add the contents of storage to a general register. The storage area must be 2 or 4 bytes long.
- **#ALD** Add logical double. Adds a doubleword to a general register pair. General register 15 cannot be coded for any of the operands.
- **#C** Compare a general register against storage.
- **#L** Load a general register.
- **#LT** Load a general register and test its contents.
- **#SLD** Subtract logical double. Subtracts a doubleword to a general register pair. General register 15 cannot be coded for any of the operands.
#ST Store the contents of a general register in storage.

## Miscellaneous Generalised Instructions

### INCR - Increment

```
INCR <target>  
[<increment>]  
[REG=<register>]  
```

- **<target>** The area to increment. A general register is indicated by R followed by a number from 0 to 15. Other targets are areas in storage.
- **<increment>** How much to increment the target. One is the default.  
- **REG** Specifies a work register when a storage target is used. The default is 15.

**Note:** Incrementing is done by LA. This discards a number of leading bits, depending on the addressing mode.

### DECR - Decrement

```
DECR <target>  
[<decrement>]  
[REG=<register>]  
```

- **<target>** The area to decrement. A general register is indicated by R followed by a number from 0 to 15. Other targets are areas in storage. 
- **<decrement>** How much to decrement the target. One is the default.
- **REG** Specifies a work register when a storage target is used. The default is 15.

Decrement of literal 1, 2, and 3 are done by BCTR; other decrements (also labels equated to 1 through 3) are done with S. Though predictable, the condition code should be considered unspecified. Use S when you wish to decrement a register and test the result at the same time.

### ZIP - Clear Registers

```
ZIP <list>  
```

- **<list> ::= <register> | <register>,<list>**  

The registers are cleared by logically subtracting them from self (SLR). The condition code is 2.
#LAL - Load Address and Length into Register Pair

```
#LAL <register>,<target>
<target> ::= <label> | '<string>' | C'<string>' | =C'<string>'
```

The address of the target is loaded in the first register in the pair (the one mentioned in the first operand). The length is loaded into the other register.

The length of a label is determined by the length attribute; the length of a literal is determined by the count of characters; the literal should not contain ampersands or quotes since they distort the count.

GENOFS - Generate Offsets to Labels in a DSECT

Use GENOFS to generate labels of the form 0#<label> for labels in a DSECT.

```
GENOFS <labels>
<labels> ::= <label> | <label>,<labels>
```

For each positional operand, a label is generated with a prefix of 0#. The new label is equated to the difference between the label coded and the beginning of the current CSECT or DSECT. The macro is most useful when coded in a DSECT.
Chapter 8. Writing CMS/TSO Pipelines Stages in Structured Assembler

Because REXX programs are so easy to write and debug, REXX is the language of choice for CMS/TSO Pipelines filters. However, a filter that is used often to process large amounts of data may not perform satisfactorily when it is written in REXX. Compiling the filter is likely to improve performance only if the filter does considerable number crunching. If the performance of the filter is dominated by moving records between REXX variables and the pipeline, compiling is unlikely to improve performance.

Thus, you may wish to write filters in assembler to improve performance. You must also write a filter in assembler if it must start at a commit level below -127 or if it uses interfaces that are not available in REXX.

There are three variants of the interface between an assembler program and CMS/TSO Pipelines. The three variants differ in the entry conditions; all dispatcher services are available to programs independent of the entry conditions they require. The three interfaces are:

- The original interface published in CMS Pipelines Toolsmith's Guide and Filter Programming Reference, SB11-6605 (SL26-0020 in the USA and other countries). (The “vanilla” interface.) This interface tries to mimic the entry conditions for a CMS command. It provides tokenised and untokenised parameter lists. Additional information is provided in general registers 3 through 6. The intent was to allow for dual-use programs that could determine dynamically if they were invoked as CMS commands or as filters, but this never caught on. This interface is now archaic; we do not recommend its use for new development.

- The program descriptor was introduced in modification level 3 to ease the migration to a 31-bit interface and to allow the scanner to perform some syntax checks. The procedure work area is pre-allocated by the scanner. The address and length of the argument string is passed directly in a register pair. This interface (the “mocca” interface) is declared by a PIPDESC macro that does not include the HLL= operand.

- The “spicy” interface is used when the program descriptor includes the HLL=ASM operand. An argument structure is passed to the stage in general register 1; the address of the work area is passed in the forward pointer slot of the save area. Other registers are as for the “mocca” interface. You must use the argument structure to issue requests for storage below the 16M line. (Though you could build an arguments structure and pass it to CMS/TSO Pipelines using any of the three sets of entry conditions, it is easier to use the argument structure passed to the stage on initial entry.)

CMS/TSO Pipelines filters and other built-in programs are written as procedures. The entry point that is referenced from the entry point table is the label on a PIPDESC macro rather than an executable instruction. The PIPDESC macro expands to a program descriptor. This is a control block that describes the program to CMS/TSO Pipelines. There are many advantages to using a program descriptor:

- The procedure work area is pre-allocated by the pipeline scanner. The scanner allocates work areas contiguously for all programs that are invoked through a program descriptor; this can reduce storage management overhead. This may also improve the locality of the working set.
The scanner can verify that the stage is (or is not) a first stage; that it has (or has not) a non-blank argument string; that it has no more than a specified maximum number of streams defined.

The scanner can “run” the syntax program and call the syntax exit to ensure that the program’s argument string is syntactically correct.

Buffer structures (declared by PIPBFR macros), are automatically de-allocated (released) when the stage terminates.

If you write your filter with the macros to define procedures, you too can use PIPDESC to declare the entry point. This chapter deals with procedures using PIPDESC to define the stage.

Many worries go away when you use PIPDESC to declare the entry point of a stage:

- No flag byte is passed in the high-order byte of register 1; the argument string and the work area are in general allocated above the 16M line. No tokenised parameter list is built; no storage is required below the 16M line.

- CMS/TSO Pipelines infrastructure allocates the work area. Specify FREETYPE=NONE on MODBEG. This means that the module does not issue storage management requests directly; thus it can run on any operating system that is supported by CMS/TSO Pipelines (assuming, of course, that the program does not issue CMS macros).

- You can check the syntax of the arguments before any stages are run. The scanner aborts the complete pipeline if any errors are found (that is, if you return with a non-zero return code).

To test the filter, you can load the object module into the user area (the CMS LOAD command) and use the ldrtbls look-up routine to resolve the entry point. For production, your program should be part of a filter package and be identified in the entry point table in the filter package.

When CMS/TSO Pipelines has found your filter’s PIPDESC macro, it invokes the procedure bodies with a pre-allocated work area and with specified information in the general register set. General register 9 contains the address of the pipeline services transfer vector (PSTV), which is a list of entry points to CMS/TSO Pipelines services. (This address is also in general register 5.) You must guard this address well; if you lose it, you will not be able to call on CMS/TSO Pipelines services. Keep the address in general register 9 or store it somewhere. Issue PIPEPVR to declare where it is. Issue PIPEPVR to declare where it is. Use LOC= to designate the label on a word in the work area if you have stored the address in the work area (note that PIPEPVR does not store this address). Refer to CMS Pipelines: Toolsmith’s Guide, SL26-0020 for a description of the macros you can use to request services from CMS/TSO Pipelines.

When you specify HLL=ASM, your filter is called with a standard parameter list that includes the address of the entry point vector provided in general register 9 in the other interface. Use the services described in Virtual Machine CMS Pipelines: Application Development Reference[9] or use the “old” services. (You must issue PIPEPVR to declare where the Pipeline services transfer vector is.)
Procedure Bodies in a CMS/TSO Pipelines Stage

Because the scanner allocates the procedure work area, you must specify $AREA=STACK on the PROC macro for the main procedure (assuming you are using the “mocca” entry conditions). If you use the “spicy” entry conditions, you must specify $AREA=HLLSTACK. You should also specify ENTRY=NO, because the procedure is reached via the program descriptor; there is no external reference to the procedure’s entry point.

The procedure body in which the PIPDESC macro is specified is invoked when the stage starts running. In addition, you can declare the name of another procedure body to be called as part of the syntax check. This procedure body is called the syntax exit. The two procedure bodies must be part of the same main procedure; they share the work area. The work area is left unchanged from the time the syntax exit returns until the stage is started.

The scanner calls all syntax exits before it passes the pipeline specification to the dispatcher to be run. If all syntax exits return 0 in general register 15, the main procedures are called to run the pipeline; the pipeline specification is abandoned if any syntax exit returns with a non-zero return code or if any syntax program (see below) issues an error message.

The Syntax Exit

You can check the syntax of the arguments before the pipeline is run. The syntax of the stage is defined by one or more of these:

- Operands on the PIPDESC macro indicate over-all attributes, for example that the stage must have arguments.
- The SYNTAX= operand. The operand field specifies the address of a syntax program or (if it begins with a left parenthesis) contains the syntax program.
- The syntax exit.

The syntax operand supports a convoluted language described in “Coding a Syntax Program” on page 60. You may find it difficult to understand; instead, you can specify the SYNTROUT= operand and omit the SYNTAX= operand. This will cause the scanner to call the specified procedure body to perform the syntax check. When the syntax exit is entered, general registers 2 and 3 contain the address and length of the parameters; you are all set to use the scanning routines. You should check that there are no residual arguments after you have scanned what you want; issue message 112 when you see more arguments than you require. Never ignore excess arguments “quietly”; as often as not they represent a new stage where the stage separator has been deleted.

Alternatively, you can call your syntax exit with this specification:

SYNTAX=(CALLSYNT,DONE)

This calls the procedure with the entry point specified by the SYNTROUT= operand. The syntax exit should pass the remaining argument string back in general registers 2 and 3. When the syntax exit returns, DONE tests if there are more arguments and issues message 112 for the excess parameters.

Registers and Syntax Programs: The scanner creates an initial register set that contains the values specified below. The contents of a register in the register set can be modified by storing into the contents of a register in the syntax program (using the slash operator). This updated value will be passed to the main entry. (The syntax program can even store into the register set after the syntax exit has been called.) If the syntax program does not store into the register set, the register contents are the same at the beginning of the syntax.
exit and at the main entry. If the syntax exit passes values back in registers, these modified values will be seen by the remaining syntax program, but they will not be seen by the main entry unless they are stored into the register set by the syntax program.

**Registers at the Beginning of a Syntax Program:** The registers at entry to the syntax exit are as initially set up, if the syntax program has CALLSYNT as the first operation.

- **R0** New-format parameter list for verb and arguments, or the address of the SCBLOCK for the EXECCOMM when GBL=FPGXCOM is coded.
- **R1** Address of work area to be used (HLL=NO); address of the parameter list if HLL=ASM.
- **R2** Address of the parameter string. Return the address of unscanned parameters.
- **R3** Length of the parameter string. Return the length of the unscanned parameter string.
- **R4** Number of highest-numbered stream.
- **R5** Pipeline services transfer vector.
- **R6** Pointer to the save area where the register set for the main entry are stored. This register must be returned unchanged. (Note that this register contains the message level when the main entry is called.)
- **R7** This register must be returned unchanged.
- **R8** This register must be returned unchanged. (It contains the address of the next syntax instruction to issue.)
- **R9** Pipeline services transfer vector (also in general register 5). This register must be returned unchanged.
- **R10** Address of the program descriptor. This register must be returned unchanged.
- **R11** Pointer to the save area where the register set for the main entry is stored (also in general register 6.) This register must be returned unchanged.
- **R12** Entry address.
- **R13** 18-word save area.
- **R14** Return address.
- **R15** At the beginning of the syntax program general register 15 is zero if the stage if a first stage and positive otherwise. When calling the syntax exit general register 15 has the entry address. Return zero if the syntax is correct and the pipeline should proceed. The pipeline is abandoned if the return code is not zero.

Red Neon!

**Note:** Only PIPERM and the macros described in *Macros for Argument Scanning and Conversion* in *CMS Pipelines: Toolsmith’s Guide*, SL26-0020, are allowed in a syntax exit. These functions are listed as conversion routines in *Virtual Machine CMS Pipelines: Application Development Reference*. Results are unpredictable, but invariably unpleasant, if you use any other CMS/TSO Pipelines function. Only change general registers 6 through 14 if they are saved and restored.

### Main Procedure

The main procedure is declared implicitly as the procedure body after which PIPDESC is coded. Thus, you should issue the macro in the static data area after the procedure body you want invoked as the main entry for the stage; there can be more procedure bodies after the PIPDESC macro. (And for that, there can be several PIPDESC macros for one procedure body.)

When a syntax exit has been used, you should specify INITWA=NO on the PBEGIN for the main procedure body. This will retain the values initialised in the syntax exit. This also applies to implicit initialisation in FLAGDEF.
Registers at Main Entry: The initial contents of the stage’s register set is shown below; the registers can be changed by a syntax program, and in a syntax exit.

R0 New-format parameter list that contains pointers to the beginning of the verb and the arguments. This information is already provided in general registers 2 and 3; there is little need to inspect this New-format parameter list. But is should be left undisturbed, because it is used by the message service when it issues message 1 to identify the stage that issued a message.
R1 Address of work area to be used (HLL=NO); address of the parameter list if HLL=ASM.
R2 Address of the parameter string.
R3 Length of the parameter string.
R4 Number of highest-numbered stream.
R5 Pipeline services transfer vector.
R6 Message level.
R7 Address of the user flags. The fullword pointed to is in the format of the field PIPARGLOCALOPTIONS in the argument structure.
R8 Stage number (general register 3 in the standard interface).
R9 Pipeline services transfer vector (also in general register 5).
R10 Address of the PIPDESC macro expansion; the program descriptor.
R11 Unspecified. (Contains the user flags.)
R12 Entry address.
R13 24-word save area. The call type flags follow this area.
R14 Return address.
R15 Entry address. Return the stage’s return code.

Defining the Stage

The label on PIPDESC is the entry point to be coded in the entry point table. Code like this to invoke the sample in “Sample CMS/TSO Pipelines Stage” on page 58 as either tokenise or tokenize. The asterisk marks the beginning of comments. Do not code a fourth token of the entry; the language is implied by PIPDESC and should not be coded explicitly.

```
tokenise       PIPTKNIS  * Tokenise
tokenize       PIPTKNIS  * Tokenise
```


PIPDESC - Declare a Pipeline Stage
ENTRY Code NO to suppress an entry statement for this table; the table is identified in a local entry table in the module.

SYSTEM List the host system(s) where this stage is supported. The default is not to check for the host system. These values can be coded:

- **CMS** CMS in a 370-mode virtual machine.
- **CMSXA** CMS 5.5 (or later) in an XA-mode (ESA-mode or XC-mode) virtual machine.
- **MVS** MVS/XA and MVS/ESA.
- **GCS** Group Control System in a 370-mode or XA-mode virtual machine.

**Note:** The fact that you can code environments other than supported ones does not mean that *CMS/TSO Pipelines* can be installed in such an environment, nor that *CMS/TSO Pipelines* works in this environment, nor that there is any intent to make it work.

HLL Code HLL=ASM when you wish to use the interface in *Virtual Machine CMS Pipelines: Application Development Reference*. You must use this interface to be able to allocate storage for buffers below the 16M line.

ABOVE Code NO if the work area must be below the 16M line in an XA-mode virtual machine. The work area goes above the 16M line by default.

STREAMS Code the maximum number of streams supported by the stage. Message 264 is issued if the stage has more streams defined. *CMS/TSO Pipelines* ensures that no more streams are defined, but it does not enforce a minimum. Test the value in general register 4 to ensure a minimum number of streams. Message 222 is recommended when there are too few.

FIRST Specify YES. if this program must run as a first stage. Specify NO. if it must not run as a first stage. Omit this operand if the program can run anywhere in a pipeline.

ARGS Specify YES. if this program must receive a non-blank parameter list. Specify NO. if it must not have arguments. Omit this operand if arguments are optional for the program.
FP Specify NO if the program uses a program mask of zero and it does not require the floating point registers to be maintained across calls to CMS/TSO Pipelines services. This operand reduces the dispatcher overhead significantly.

SYNTAX Code parentheses with the syntax definition. This can be quite complicated; the easiest is to code like this to call the entry point to perform the syntax check: SYNTAX=(CALLSYNT,DONE).

SYNTROUT The name of the procedure body (EP= on PBEGIN) you wish to have called to perform the syntax check.

COMMIT The commit level at which the program can start. The default is 0. CMS/TSO Pipelines built-in programs run on commit level 0. The number is normally negative.

EXIT Specify the address of the dispatcher exit that applies to the stage. The dispatcher exit can also be set or retracted with the macro PIPEXIT.

BUFFER Specify the label on a PIPBFR macro in the work area. The buffer is deallocated automatically when the stage returns; do not write a PIPFRBF macro to return the buffer explicitly. Specify multiple buffers as a sub-operand list.

STOPABLE Write YES if the dispatcher can summarily terminate the stage. This is the case then the stage has allocated no resources (possibly except for buffers specified by BUFFER=) and the stage should be terminated when all output streams are not connected or all input stream are at end-of-file. RC8 means that the dispatcher returns code 8 on a PIPLOCAT when all output streams become not connected while the stage is waiting for input in a PIPLOCAT macro (or similar).

The following operands are obsolete, but may still be found in CMS/TSO Pipelines code:

FLAGS Four bytes of flags can be coded in sublists in parentheses. Use plus (+) to combine flags within a particular byte. No parentheses are needed to specify flags in the first byte only. It is not verified that the flag is turned on in the correct byte. These flags are defined in the first byte:

FPNOARGS No arguments are allowed; message 112 is issued if the argument string is not blank.

FPMUSTARG There must be a non-blank argument string; message 11 is issued if none is found.

FPFIRST The stage must be first in a pipeline specification; message 87 is issued if it is not.

FPNFIRST The stage must not be first in a pipeline specification. message 127 is issued if it is.

FPNOFPR Floating point registers are not used or their integrity needs not be maintained across calls to the pipeline dispatcher. The program mask is zero. Code this flag to improve dispatcher performance.

GBL Flags reserved for CMS/TSO Pipelines. The pipeline specification scanner combines the global flags for all stages in a pipeline by an inclusive-or operation.

FPGXCOM Store the address of the active &ix when the scanner is invoked. This address is made available to the stage in register 0.
Coding an Entry Point Table as Constants

The entry point table is normally a separate file, from which a utility generates an object module which you include when you build the MODULE file for the filter package. However, you can assemble an entry point table into the program:

- As a convenience to avoid the separate source file, for instance when the module only has one user program which has all the filters. The entry point table must have the name PIPLOCAT; this is the label of the table defining all filters in a filter package.

- When you wish filters to have names with two or more words where the first word is a generic descriptor for the set of filters. The first component is put in the entry point table for the filter package; it refers to a magic constant (X'008597A3' or X'00',C'ep') in front of the entry point table that resolves the second component of the name. There is no limit to the nesting of entry point tables.

```
ENTRY PIPEPT
  PIPEPT DS 0D
  PIPEPT
  PIPEPTEN MYPROG,GOPGM
  PIPEPTED
  SPACE 1
  CNOP 4,8 Align
  GOPGM DC A(C'ep')
  PIPEPT
  PIPEPTEN DECODE,DCODE,MIN=3
  PIPEPTEN ENCODE,NCODE,MIN=3
  PIPEPTED

Figure 19. Coding Nested Entry Point Tables
```

Figure 19 shows how to code both types of entry point tables. Note the alignment of the magic token; it must be 4 off a doubleword boundary to be adjacent to the table itself.

**PIPEPT - Define an Entry Point Table**

```
<label> DS 0D
PIPEPT

Code PIPEPT to open an entry point table. The table is aligned on doubleword boundary. PIPEPT ignores the label field; when needed, code a label on a DS0D instruction in front of PIPEPT.
```

**PIPEPTEN - Entry in an Entry Point Table**

```
PIPEPTEN <name>[,<entry>]
  [ATYPE={A|V}]
  [MIN=<number>]
  [COMMIT=<number>]
  [PGMLANG={ASSEMBLER|PL/I|IBMC370|MINIMALC}]
```

Chapter 8. Writing CMS/TSO Pipelines Stages in Structured Assembler 57
<name> The name of the filter. This is the name you code in the pipeline specification. The name is truncated after eight characters. PIPEPTEN macros must be in ascending order of the names.

<entry> The label on the PIPDESC macro defining the entry point, or the label on an entry point table identifier when referring to an entry point table to decode the next word. <name> is used as the entry point when you do not code the second positional operand.

ATYPE Code V to generate a V-type address constant to point to the entry. This may be needed when the entry point is in another module. The default is to generate a normal address constant.

MIN The minimum number of characters in the abbreviation for the entry point name. The default is no abbreviation. Always code MIN=8 when the name is longer than eight characters.

COMMIT Specify the starting commit level for the stage. The number can be from -128 to 127, inclusive. Note that positive numbers are incompatible with CMS/TSO Pipelines built-in programs.

PGMLANG Specify the programming language interface to be used. IBM370 is not recommended, because there is no documented interface to run such programs as co-routines.

**PIPEPTED - End of an Entry Point Table**

```
PIPEPTED
```

This macro closes the table and generates equates for constants in the table header.

---

**Sample CMS/TSO Pipelines Stage**

```
TKN TITLE ' Tokenise * 
COPY COPYRIGHT IBM Danmark A/S'

*********** Tokenise <dstring> [<dstring>] ***********
* 
* A line is written for each token. Blanks always separate * 
* tokens. Characters in the first delimited string are always * 
* separate tokens, for instance /() in CMS scanning rules. * 
* Second string, if present is written after each input line * 
* is processed (newline sequence). * 
* Change activity: * 
* 15 Nov 1989  New module * 
* 
***********

PIPTKN MODBEG FREETYPE=NONE 
PIPEPVR R9 
TOKENISE PROC SAREA=STACK,ENTRY=NO 
TABLE DS CL256
```
PBEGIN ,
XC TABLE, TABLE Assume not delimiters
MVI TABLE+C ',', 1 Indicate blank
LA R0, 2 What to store
ZIP R2 Clear for index
LTR R5, R5 Test length of string1
REPEAT COND=POSITIVE
  IC R2, 0, (R4) Get char
  STC R0, TABLE (R2) Make delimiter
UNTIL INCR=R4, BCT=R5 Loop through the lot
EJECT ,
***********************************************************************
* *
* Process input records. *
* *
***********************************************************************
SPACE 2
REPEAT ,
  PIPLOCAT EXIT=NEGATIVE
  GOTOEXIT 0, COND=NOTZERO
  LR R2, R1 Get base
  LTR R3, R0 And length
  * R1 is token begin
  REPEAT COND=POSITIVE
    ZIP R8 For IC
    IC R8, 0, (R2) Get char
    IC R8, TABLE (R8)
    DECR R8 Back down
    LTR R8, R8
    CONTINUE COND=MINUS Char in token
    LR R0, R2
    SR R0, R1
    IF POSITIVE
      PIPOUTP EXIT=NOTZERO
    FI
    LTR R8, R8 Delimiter in itself?
    IF POSITIVE
      PIPOUTP ((R2), 1), EXIT=NOTZERO Write if so
    FI
    LA R1, 1, (R2) point to next
  UNTIL INCR=R2, BCT=R3
  LR R0, R2
  SR R0, R1
  IF POSITIVE
    PIPOUTP EXIT=NOTZERO
  FI
  LTR R0, R7 Any EOR to dump?
  IF NOTMINUS
    LR R1, R6 Get begin
    PIPOUTP EXIT=NOTZERO
  FI
  ZIP R0 Drop line
  PIPINPUT ,
UNTIL NOTZERO
PEXIT RC=(R15)

&MODULE.IS PIPDESC FLAGS=FPNFIRST+FPMUSTARG+FPNOFPR, STREAMS=1, *
(SDEL, ?NZ60, =4, 6, =5, 7), DONE)
PROCEND TOKENISE
MODEND
Coding a Syntax Program

Though you probably never wish to code fancy SYNTAX= operands, you may find this explanation useful to understand the code in CMS/TSO Pipelines.

The syntax check is performed simulating the program of an abstract machine. The program is coded as operands to SYNTAX= and the #SYNTAX macro.

**Note:** The register conventions described in “Procedure Bodies in a CMS/TSO Pipelines Stage” on page 52 show the contents when the syntax program begins. The program can change the register contents at will. But some registers should not be changed; for example, changing general register 8 will mean that the instruction counter for the syntax program is changed. Clearly, you do so at your peril.

The Abstract Machine

The abstract machine is a stored-program single instruction stream single data stream non-vonNeuman machine (you cannot modify the instruction stream). It has:

- A location counter: the address of the current instruction. Instructions are operands of the SYNTAX= operand of the PIPDESC macro or operands of the #SYNTAX macro.
- A register set: registers 0 through 6, and 15. The machine starts out with registers 0 through 6 as described in “The Syntax Exit” on page 52. Register 15 contains the stage’s number relative to zero (a first stage has zero in general register 15). You may modify registers 0 through 5 and 15. The machine allows access to registers 6 through 14 as well. Results are unpredictable if you modify any of the registers 6 through 14.
- A condition code: the contents of register 15. Register 15 is set to the return code when a scanning service returns to the abstract machine. The condition code can be tested for minus, zero, and positive. (The System/370 condition code is not available to the abstract machine.)
- The stage’s register set: the 24-word save area where the register set for the stage is kept in the OS format. Register 0 refers to offset 20 (decimal); the six extra words are logically referenced as registers 13 through 18, but the program in the abstract machine cannot refer to registers beyond 15. (Note that the stage’s registers 14 and 15 cannot be referenced.) The “extension registers” are used by CMS/TSO Pipelines in this way:
  - Registers 13 and 14 are the token used implicitly by UTKN and MTKN.
  - Register 15 holds the return address after a call (the minus instruction).
  - Register 18 holds a pointer to a save area that must not be lost.
- Data storage: you can store a register into the first 1024 bytes of the stage’s work area, but you cannot load a register from there; the area contains write-only storage.
- Read-only storage whence constants can be loaded into a register using the colon instruction.

The operation codes for the abstract machine are special characters, names of conversion routines (omitting the PIP prefix), and some keywords. The instruction set for the abstract machine includes instructions to:

- Branch unconditionally to a #SYNTAX macro.
- Call a #SYNTAX macro, and return. There is only one return register; calls cannot be nested.
• Test the contents of general register 15 and execute one of two expressions.
• Stop processing.
• Issue an error message and stop if a specified condition holds.
• Call CMS/TSO Pipelines scanning routines.
• Store the contents of registers into the stage’s register set or the work area.
• Load a register with a literal or the contents of a register in the stage’s register set.
• Perform register-to-register operations against general register 15.
• Perform miscellaneous specialised operations.

Sample Syntax Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>What it Does</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORD</td>
<td>Call PIPWORD to scan a blank-delimited word from the argument string in registers 2 and 3. The address and length of the word are stored in general registers 4 and 5; general register 15 is set to the length of the word; it is zero when there are no further arguments.</td>
</tr>
<tr>
<td>?Z113</td>
<td>Issue message 113 (missing parameter) if general register 15 is zero. This is done after a word is scanned when it is not an optional argument. (WORDM performs this function in a single operation.)</td>
</tr>
<tr>
<td>WORD,(?Z,(LDKWD,DP))</td>
<td>Scan for a word and use the default keyword DP if there are no further arguments.</td>
</tr>
<tr>
<td>WORD,?NZ113,DECWD,?NZ58</td>
<td>Scan a word and ensure it is present as shown above. Convert the word from decimal to binary and issue message 58 if the conversion fails.</td>
</tr>
<tr>
<td>=4,MAXSTREAM,WORD,=4,4,=5,5</td>
<td>Store general register 4 at the label MAXSTREAM in the work area; then scan a word and store the address and length in registers 4 and 5 of the stage’s register set.</td>
</tr>
<tr>
<td>0TO15,NP66</td>
<td>Load the contents of general register 0 into general register 15 and issue message 66 if the word is not positive.</td>
</tr>
<tr>
<td>UTKN,.SY,?Z,(&lt;x1&gt;),(&lt;x2&gt;)</td>
<td>Load the word described by registers 4 and 5 into a doubleword token, converted to uppercase. Then test if the token is an abbreviation of the keyword with label SY in the keyword table. Process expression 1 if the word is an abbreviation of the keyword; process expression 2 if not.</td>
</tr>
<tr>
<td>:15,1,(</td>
<td>4,SR),?P222</td>
</tr>
<tr>
<td>R1,FNAME,WORDM,(CVT,UTKN),</td>
<td>Load two tokens into the doublewords beginning FNAME. Issue message 113 if there are less than two tokens in the argument string.</td>
</tr>
</tbody>
</table>

Coding a Program for the Syntax Machine

The program is coded as positional sub-operands of SYNTAX= of PIPDESC, or as positional operands of #SYNTAX. Many operations require only a single operand, for instance DONE. Operations with parameters are coded as several operands, or as a single operand with a leading special character.
**Note:** The last operand of a syntax operand list must transfer control unconditionally. Do not expect control to flow sequentially from one #SYNTAX macro to the next. Use DONE, STOP, RETURN, ;<label>, or <number>.

**Control Flow:** These prefix characters represent control flow operations:

- **;** (Semicolon) Unconditional branch. Write the label of a #SYNTAX macro after the semicolon.
- **-** (Hyphen) Call a subroutine. Write the label of a #SYNTAX macro after the hyphen. The RETURN instruction returns control to the following instruction. The subroutine must be within 512 bytes (either direction) of the call. Use the CALL instruction to call a subroutine that is further away or in another module.

**Note:** Calls cannot be nested; there is only one return address register.

- **?** (Question mark) Test the condition code in general register 15. The condition code to select is specified in the following mask characters. It can be Z, P, M, NZ, NP, NM.

  Code a number after the condition code to issue this message and terminate the syntax machine if the condition is true.

  When there is no number after the condition code, the next operand or the following one is executed, depending on the relation between the contents of general register 15 and the mask characters. Code expressions in parentheses when more than one operation is required in an alternative. Code the test and a single action (possibly a sublist) in parentheses to elide the alternative.

- **<number>** Issue the message of that number and terminate the syntax program.

These control-flow operations are keywords:

- **RETURN** Return to the instruction following the previous call.
- **DONE** Ensure that only blanks remain in the argument string. Terminate syntax scan with message 112 if a non-blank argument is found. Stop processing with return code zero if there are no further arguments.
- **STOP** Processing stops without testing for the argument string being exhausted; the return code is zero.
- **DONEZ** Stop processing if no more arguments remain in the string described by general registers 2 and 3. This is a convenience for STRLB,(!3,LR),(?Z,DONE).

**Calling Scanning Routines:** The scanning routines act on general registers 0 through 5. Refer to CMS Pipelines: Toolsmith’s Guide, SL26-0020 for a description of each.

Code the name of the scanning routine without the leading PIP, for instance WORD. When calling routines UTKN and MTKN, general register 1 is first loaded with the address of a doubleword after the stage’s register set. To load a token into a doubleword in the work area, use the R1 operation followed by an explicit CVT to request the conversion without loading general register 1.

Instead of TSTKW, code a period before the symbolic name of an entry in the keyword table to test if the token addressed by general register 1 is an abbreviation of this keyword. General register 15 is set to zero if it is, and to 1 otherwise.
**Store the Contents of a Register:** A leading equal sign requests that a register be stored. The rest of the operand specifies the register number. The next operand specifies the destination.

= (Equal sign) Store the contents of a register. When the next operand is numeric, the register is stored into the corresponding register of the stage’s register set. The register is stored into the work area when the following operand contains a label.

**Load a Register:** A leading special character indicates the type of source; the destination register number is coded after this. The source of the load is coded in the following operand.

: (Colon) Load a literal into a register. The literal is coded in the following operand; it must be suitable for a length-4 address constant.

/ (Slash) Load a register from the stage’s register set. The following operand is the register number to load from.

**Perform a Register Operation on General Register 15:** Code an exclamation sign followed by the register number. The following operand is the operation code to perform.

**Miscellaneous Special Operations:** The special-purpose operations are:

- **0TO15** Copy general register 0 to general register 15 where it can be tested. A convenience for \( (10,L) \).

- **WORDM** Scan a word and issue message 113 if there are no more arguments. A convenience for \( \text{(WORD,} Z113) \).

- **CALLSYNT** Call the procedure body in \( \text{SYNTROUT=} \). The syntax machine stops if the return code is non-zero. A non-zero return code also causes the pipeline to be abandoned; failure exits are called for stages where the syntax exit returns zero when the pipeline is abandoned.

- **STR16M** Ensure that the contents of general register 5 is less than 16M (the high-order byte must be zero); issue message 284 if this is not the case.

- **TST112** Issue message 112 if there are more arguments.

- **R1** Load general register 1 with the address of a label in the work area.

- **LDKWD** The following operand specifies the two-character keyword label. For the program offering, PIPKWD is prefixed to generate the external reference to the keyword table. VM/ESA uses DMSPRM as the prefix for keywords.

- **CVT** Explicit call to conversion routine. Used with MTKN and UTKN to load 8-byte tokens into the work area. The next operand is the name of the conversion routine to call.

- **INCR1** Increment the contents of general register 1. The following operand specifies the increment.

- **CALL** A call to a \#SYNTAX macro in a different module or more than 512 bytes away from the call. The next operand specifies the label of the \#SYNTAX macro to call. The return address is shared between the minus instruction and this call instruction; only one level of call is possible unless you save and restore the contents of extension register 15.
Chapter 9. Assembler Errors and MNOTEs

This chapter lists the MNOTEs issued when errors are detected by the macros. MNOTEs are listed in the order of their identifiers.

Fix-up after errors in the procedure and flow control macros is not always good. A cascade of addressability errors can occur as a result of some errors.

In such a situation, fix the first few errors and inspect the end of the error listing for trouble with the nesting of procedures and correct any problems there. Then re-run the assembly.

Errors Reported by the Assembler

Addressability

If the pro/epilogue for a procedure encounters addressability errors for a literal, it may help to code an explicit $EP=$ to generate the code in-line. PLTORG generates a literal pool after the current procedure.

Addressing errors for PCALLs to level-1 procedures are often fixed by moving the called procedure to the beginning of the module; then PCALL only needs addressability to the literal pool with the address of the routine to be called.

Addressing errors are likely to result if the procedure body is larger than 4K. Split the procedure into smaller ones.

Undefined Symbols

Some macros ignore the contents of the label field. This can lead to undefined symbol diagnostics.

Invalid Duplication Factor

Trouble with a CASITEM indicates that the item is out of the range declared on CASBEG.

Trouble with syntax programs indicate that the offset is too large for the number of bits available; move #SYNTAX macros around, ensure that you only store into fields in the first 1024 bytes of the work area.
### INDC001E Increment/decrement is 0

**Explanation:** The second operand of an INCR or a DECR macro is the literal 0.

**Severity:** 4

**System Action:** The requested instruction is generated.

**User Response:** The instruction is redundant. Most often the 0 is keyed from habit of using BCTR with zero target to decrement.

### INDC002E &TYPE: &WORD is not defined.

**Explanation:** The item to increment or decrement is not known by the assembler at this point in the program; its length cannot be determined and it is not possible to expand the macro.

**Severity:** 1

**System Action:** The macro is ignored.

**User Response:** Ensure that the first operand is spelled correctly and is defined before it is used. You cannot use INCR and DECR on an item that is defined in a procedure work area later in the program.

### ALD001E Specify a register and an operand

**Explanation:** Too few operands are coded.

**Severity:** 8

**System Action:** The macro is ignored.

### CKPOS001E Label "&XL" ignored.

**Explanation:** A macro has a label which is ignored.

**Severity:** 1

**System Action:** The label field is ignored.

**User Response:** Code an explicit DS instruction if you wish a label at this point in the program.

### CKPOS002E "$&X" is an excessive positional operand.

**Explanation:** More positional operands are coded than the macro expects.

**Severity:** 1

**System Action:** Additional operands are ignored.

**User Response:** Ensure that sublists are properly enclosed in parenthesis.

### COM001E &WORD is not defined.

**Explanation:** A generalised instruction refers to the word shown, but the label is not yet defined to the assembler.

**Severity:** 1

**System Action:** The macro is ignored.

**User Response:** Ensure that the name is coded correctly and that it is defined before it is referenced. Code the appropriate instructions when referring to the procedure work area of a procedure that cannot be declared before the current one.

### COM002E &WORD has length &LEN, I can only handle 1..4

**Explanation:** The label is defined, but its length attribute is not between one and four.

**Severity:** 1

**System Action:** The macro is ignored.

**User Response:** Ensure that you are referencing the correct address. Use #ALD and #SLD to perform doubleword integer addition and subtraction.

### COM003E Please specify a register and an operand.

**Explanation:** Too few operands are coded.

**Severity:** 4

**System Action:** The macro is ignored.

### COM004E I am sorry, but I cannot generate for this length.

**Explanation:** The generalised instruction cannot be generated for the operand with the length it has.

**Severity:** 4

**System Action:** The macro is ignored.

**User Response:** Code using the standard instruction set if the length of the operand is indeed what you desire. For instance, the macro cannot add a three-byte field to a register.

### DCDMA001E Unknown mask &C

**Explanation:** The condition code is not known for the operand shown.

**Severity:** 8

**System Action:** The mask is ignored.

**User Response:** You cannot use your own equates for the mask values.
#ELSE.001E &DUPMSG already defined for this level
Explanation: An unexpected ELSE is met.
Severity: 12
System Action: The macro is ignored.
User Response: Correct the error in the structure of the program.

#FLAGM001E Operand(s) &S ignored
Explanation: More than one positional operand is found.
Severity: 4
System Action: Remaining positional operands are ignored.
User Response: Enclose a list of flags in parenthesis so that it becomes a single positional operand.

#FLAGM002E &FX is not a flag.
Explanation: The operand shown is not declared with FLAGDEF.
Severity: 8
System Action: The flag is ignored.
User Response: Flags must be declared before they can be used. If you are referring to a flag that is declared in a procedure work area in a procedure that cannot be moved in front of the current one, then put a label on the FLAGDEF and use standard machine instructions to test the flag in the flag byte.

#FLAGM003E Flag &FX is in &FN., and &F1 is in &FN1.
Explanation: A number of flags are manipulated, but they are not in the same flag byte.
Severity: 8
System Action: The offending flags are ignored.
User Response: Use FWDPROC at the beginning of the module to declare that the procedure requires a work area provided by the caller.

#LALMA001E Excessive positional operand &JUNK
Explanation: More than two positional operands are found.
Severity: 8
System Action: The additional positional operands are ignored.

#LEACO001E No matching repeat found.
Explanation: No REPEAT statement is found with the identifier coded on a CONTINUE or LEAVE macro.
Severity: 8
System Action: The macro is ignored.

#PCALL001E Unrecognised function code "&FUN."
Explanation: Internal error in the macro library.
Severity: 8
System Action: The function is ignored.
User Response: Report the problem to your system support staff.

#PCALL002E Call to procedure &PNAME(&PCURR) not in its lexical path
Explanation: A nested procedure is called from a procedure that does not contain it.
Severity: 12
System Action: Depends on when the error is found. A BAS may have been generated for a forward reference or the PCALL macro may be ignored.

#PCALL003E Forward call to &PNAME(&PCURR) not generated for SAREA=STACK. Last call from &PNAME(&FROM).
Explanation: A forward reference to the procedure was generated without knowing that the procedure requires the address of a work area in general register 1.
Severity: 12
System Action: None.
User Response: Use FWDPROC at the beginning of the module to declare that the procedure requires a work area provided by the caller.

#PCALL004E Procedure &PNAME(&J) declared not to call further
Explanation: A PCALL is found in a procedure declared SAREA=NO and SAVE=NO.
Severity: 8
System Action: The macro is ignored.

#PCOMP004E No calls to this procedure
Explanation: A nested procedure is met that has no calls.
Severity: 8
System Action: The macro is partly ignored. A large number of errors are likely in the rest of the module; they should be ignored.
User Response: The most likely cause is a misspelled name either in the label field of the PROC instruction or in the PCALL to the procedure. Code PCALL GEN=NO to declare the call if you call the procedure by other means, for instance a branch table.

#PDEF.001E Unrecognised function &FUN
Explanation: Internal coding error in the macro library.
Severity: 8
System Action: Depends.
User Response: Report the error to your system support staff.

#PDEF.002E Procedure Pipeline defined twice. Request is &TYPE., but it is already &PTYPE(&PIX).,
Explanation: A procedure is already declared with the name shown.
Severity: 8
System Action: The macro is ignored. A large number of errors is likely in the remainder of the program; ignore them.
User Response: Code the name of the procedure correctly. Remove an EXTPROC for the procedure if one exists.

#PMARK001E #PMARKR: Nothing to mark
Explanation: Internal coding error in the macro library.
Severity: 4
System Action: Depends.
User Response: Report the error to your system support staff.

#REGCH001E "&REG" not recognised as a register
Explanation: The operand should designate a general register, but it is not an integer in the range 0 through 15 or such an integer prefixed “R”.
Severity: 8
System Action: The macro is ignored.
User Response: You cannot use your own equates for registers.

#SLD..001E Specify a register and an operand
Explanation: An insufficient number of positional operands is coded.
Severity: 8
System Action: The macro is ignored.
CASBEG003E Register not specified
Explanation: The register with the value to test is not coded.
Severity: 8
System Action: The macro is ignored.
User Response: Code a value for REG=.

CASITE001E Case item is unreachable
Explanation: A CASITEM without operands has already been reached for the CASBEG.
Severity: 4
System Action: None. The code after the current macro will be unreachable.

CASITE002E No selection possible
Explanation: The first CASITEM met has no positional operand and is taken to be the default case.
Severity: 4
System Action: None.
User Response: Code the CASITEM macros in the order you wish the tests to be carried out.

CASITE003E Default item already met for this level
Explanation: A CASITEM without operands has already been met for this selection.
Severity: 4
System Action: None. The code is unreachable.
User Response: Most likely a CASEND is missing in nested selections.

CASEXI001E No matching case statement found.
Explanation: A CASEXIT is issued outside a case structure or with an identifier which does not match the identifier of any open case structure.
Severity: 8
System Action: None.

COND..001E Too many operands; π and following are ignored.
Explanation: More than 6 positional operands are present.
Severity: 4
System Action: The excess operands are ignored.
User Response: Use IF instead; you cannot code a macro reference as a sublist.

CONDSU001E Modifier must be NOT or nothing. It is "&MOD;".
Explanation: The third operand of CONDSUP is not coded correctly.
Severity: 4
System Action: The modifier is ignored.
User Response: When a number of environments apply, they must be coded as the second positional operand in parenthesis.

EXTPRO004E Invalid save-type specification "&T".
NO/YES/STACK/HLLSTACK are valid
Severity: 8
System Action: None.
User Response: The procedure declaration is ignored.

FL....001E FI has operand &SYSLIST(1) -- Probably misspelled IF
Explanation: The FI macro found an operand.
Severity: 1
System Action: The operand is ignored.
User Response: Code a comma to indicate that no operands are present when a comment is coded on a FI macro.

FLAGDE001E Too many flags starting with &FIRST
Explanation: More than eight positional operands are coded on FLAGDEF.
Severity: 4
System Action: The remaining flags are ignored.
User Response: Code one FLAGDEF for each flag byte.

FREE..001E Storage management disabled for this module
Explanation: FREETYPE=NONE is coded on MODBEG, so it is not possible to generate code.
Severity: 8
System Action: Storage is not allocated.
User Response: Code a procedure structure that does not require free storage to be allocated or assign free storage management.

FREE..002E Clear option &CLEAR ignored
Explanation: The keyword is not coded with one of the supported options.
Severity: 4
System Action: The keyword is ignored; storage is not cleared.
Chapter 9. Assembler Errors and MNOTEs

FREE..003E Storage management for &STORMGMT not implemented
Explanation: The macro has no code to cater for the type of storage management substituted.
Severity: 8
System Action: The macro is ignored.
User Response: Correct the FREETYPE= value.

FREE..004E Not cleared, size=&SIZE, bytes=&BYTES
Explanation: The macro was unable to determine the length to clear.
Severity: 1
System Action: Storage is not cleared.

FREE..005E Not cleared; R1 not available
Explanation: BASE= must be coded with CLEAR=MVCL.
Severity: 4
System Action: Storage is not cleared.

FRET..001E Storage management disabled for this module
Explanation: FREETYPE=None is coded on MODBEG, so it is not possible to generate code.
Severity: 8
System Action: Storage is not released.
User Response: Code a procedure structure that does not require free storage to be allocated or assign free storage management.

FRET..002E Storage management for &STORMGMT not implemented
Explanation: The macro has no code to cater for the type of storage management substituted.
Severity: 8
System Action: Storage is not released.
User Response: Correct the FREETYPE= value.

FWDPRO0001E Invalid save-type specification "&T". NO/YES/STACK/HLLSTACK are valid
Severity: 8
System Action: The specification is ignored. Other positional operands are processed.

GENOFS001E Label &N not defined. Offset set to zero.
Severity: 8
User Response: Ensure the label is spelled correctly and is defined before GENOFS is issued.

GOTOEX001E Return code would upset the base register
Explanation: GOTOEXIT is coded with a positional operand for the return code. The base register for the procedure is general register 15, however, so the value cannot be loaded.
Severity: 1
System Action: The return code is ignored; general register 15 remains unchanged.
User Response: Do not code SAREA=NO and SAVE=NO when you wish to code a return code on a GOTOEXIT.

IF....001E No condition specified --- probably misspelled FI
Explanation: The first positional operand is null.
Severity: 1
System Action: The macro is ignored.
User Response: Specify a condition code or change the macro to FI.

INTPAR001E Key word &KW not recognised
Explanation: Error in PROC COPY.
Severity: 8
System Action: The macro is ignored.
User Response: Contact your system support staff.

INTPAR002E Unknown keyword value &KVAL for &KW.
Explanation: Error in PROC COPY.
Severity: 4
System Action: The macro is ignored.
User Response: Contact your system support staff.

LT....001E You don’t want to do that!
Explanation: Macro to intercept LT.
Severity: 8
System Action: The macro is ignored.
User Response: Code #LT to load and test a value.
MODBEG001E Correct form of TRANSIENT is (YES,<entry point>) or NO
Explanation: An unacceptable specification is encountered.
Severity: 8
System Action: The TRANSIENT operand is ignored.

MODEND001E Unclosed procedure &PNAME(&PCURR) on level &PLVL(&PCURR)
Explanation: There are more PROC macros than PROCEND.
Severity: 16
System Action: None.

MODEND002E &PCURR contains itself
Explanation: There is an error in PROC COPY.
Severity: 24
System Action: Processing is terminated; prologues and epilogues are not generated.
User Response: Contact your system support staff.

PBEGIN001E No active procedure declaration
Explanation: PBEGIN is met outside a procedure.
Severity: 16
System Action: The macro is ignored.

PBEGIN002E PBEGIN already coded for this procedure or PEXIT missing
Explanation: The procedure body has already been opened for this procedure.
Severity: 4
System Action: The macro is ignored.
User Response: Code a PEXIT if you wish to have multiple procedure bodies in a procedure.

PBEGIN003E PBEGIN not recognised here... Maybe PROC missing
Explanation: The previous procedure body has not been closed.
Severity: 4
System Action: The macro is ignored.
User Response: Code a PEXIT to end the procedure body.

PBEGIN004E You must specify a sub entrypoint name
Explanation: An entry point name must be coded on subsequent procedure bodies.
Severity: 4
System Action: The macro is ignored.

PBEGIN005E You must generate save area for sub-entry points
Explanation: DCL=NO is coded on PROC.
Severity: 4
System Action: The macro is ignored.

PCALL.001E No procedure declaration active
Explanation: PCALL is met outside a procedure.
Severity: 8
System Action: The macro is ignored.

PCALL.002E Entry Point declared not to have backwards references
Explanation: The procedure is declared with FWDONLY=YES.
Severity: 8
System Action: The macro is ignored.

PCMS..001E The module is being assembled for &STORMGMT., not CMS
Severity: 4
System Action: The macro is ignored.

PDC...001E PDC Must be within a procedure work area; is in &SYSECT loctr &SYSLOC
Explanation: A PDC macro is met outside the procedure work area.
Severity: 8
System Action: The macro is ignored.

PDS...001E PDS Must be within a procedure work area; is in &SYSECT loctr &SYSLOC
Explanation: A PDC macro is met outside the procedure work area.
Severity: 8
System Action: The macro is ignored.
**PDSECT001E** Bad location counter &LOCTR.. Only 0 and 1 are allowed  
**Explanation:** The section in the procedure work area is not coded correctly.  
**Severity:** 8  
**System Action:** The first part of the procedure work area is opened.

**PDSECT002E** Procedure &PNAME(&PCURR) declared not to allocate a work area  
**Explanation:** The procedure is coded SAREA=NO.  
**Severity:** 4  
**System Action:** The macro is ignored.

**PEXIT.001E** PEXIT already done for procedure  
**Severity:** 4  
**System Action:** The macro is ignored.  
**User Response:** This may be the result of trouble with PBEGIN.

**PEXIT.002E** No PBEGIN for this procedure. Code likely to be in DSECT  
**Explanation:** The procedure work area is not closed with PBEGIN.  
**Severity:** 16  
**System Action:** None.  
**User Response:** Code PBEGIN after the procedure work area or code DCL=NO to indicate that the procedure should not have a work area.

**PEXIT.003E** Invalid CC specification &CC  
**Severity:** 4  
**System Action:** The operand is ignored.

**PEXIT.004E** RC takes precedence over CC  
**Explanation:** Both operands are coded.  
**Severity:** 4  
**System Action:** The CC operand is ignored.

**PEXIT.005E** REGS=&REGS but SAVE=NO implies (R0,R11)  
**Explanation:** SAVE=NO is coded on PROC.  
**Severity:** 1  
**System Action:** The operand is ignored.

**PROC..002E** Bad save area specification, &SAREA; must be YES or NO  
**Explanation:** These are the only valid specifications for a nested procedure.  
**Severity:** 4  
**System Action:** SAREA=YES is assumed.  
**User Response:** Do not code SAREA=STACK for a nested procedure.

**PROC..003E** Entry specification "&ENTRY." not acceptable. Must be YES or NO.  
**Severity:** 4  
**System Action:** No entry is generated.

**PROC..004E** Invalid save-type specification "&T". NO/YES/STACK are valid  
**Explanation:** The value coded with SAREA= is not acceptable.  
**Severity:** 8  
**System Action:** A save area is allocated.

**PROCEN001E** Procedure stack underflow  
**Explanation:** There are more PROCENDs than PROCS  
**Severity:** 16  
**System Action:** The macro is ignored.

**PROCEN002E** &PNAME(&PCURR) does not match &N  
**Explanation:** The positional operand does not match the name of the procedure.  
**Severity:** 4  
**System Action:** None.

**PTR...001E** Pointer list at &N forced aligned fullword.  
**Severity:** 4

**PWRTER001E** Invalid MF &MF  
**Severity:** 8

**PWRTER002E** Invalid MF &MF  
**Severity:** 8
STORMG001E Storage mgmt &GENTYPE requested; only &REQUEST available

Explanation: A value is specified in the SYSPARM string that is not found in the FREETYPE= specification.

Severity: 8
System Action: Storage management is disabled.

UNTIL.001E Excessive positional operands: &SYSLIST(2) &SYSLIST(3)

Severity: 1
System Action: The operands are ignored.

UNTIL.002E COND and CHAIN are now semantically different

Severity: 4
System Action: None.
User Response: Code I=KNOW to indicate that you wish to test the condition code for loop termination before chasing a chain.

UNTIL.003E BXLE option ignored. BCT has precedence.

Explanation: Both operands are coded.
Severity: 4
The next page is from a program listing of a program that reads a fixed format file from disk and writes it to the pipeline.

There are two nested loops, the outer one for each input block and an inner one to write the records in the block.

The effect of the optimiser is shown in the code generated for the ELSE in statement 1508; it branches to the address 24C which is at the top of the loop.
### Figure 21. Sample Program

<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJECT CODE</th>
<th>ADDR1</th>
<th>ADDR2</th>
<th>STMT</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>000230</td>
<td>02309C5C4C6C9</td>
<td>00206</td>
<td>00207</td>
<td>1042</td>
<td>Read the records of the file and pass them on</td>
</tr>
<tr>
<td>000240</td>
<td>02407020</td>
<td>00206</td>
<td>00207</td>
<td>1044</td>
<td>Read the records</td>
</tr>
<tr>
<td>00024C</td>
<td>024C1814</td>
<td>00206</td>
<td>00207</td>
<td>1046</td>
<td>Do the lot</td>
</tr>
<tr>
<td>000262</td>
<td>02621FF</td>
<td>00206</td>
<td>00207</td>
<td>1048</td>
<td>Error</td>
</tr>
<tr>
<td>000264</td>
<td>026447800</td>
<td>00206</td>
<td>00207</td>
<td>1050</td>
<td>Point to buffer</td>
</tr>
<tr>
<td>000267</td>
<td>0267C601</td>
<td>00206</td>
<td>00207</td>
<td>1052</td>
<td>More than one record?</td>
</tr>
<tr>
<td>00026E</td>
<td>026E05C4</td>
<td>00206</td>
<td>00207</td>
<td>1054</td>
<td>No chance</td>
</tr>
<tr>
<td>00026F</td>
<td>026FF800</td>
<td>00206</td>
<td>00207</td>
<td>1056</td>
<td>Emit the record</td>
</tr>
<tr>
<td>00027B</td>
<td>027B07A</td>
<td>00206</td>
<td>00207</td>
<td>1058</td>
<td>Nothing there</td>
</tr>
<tr>
<td>00027C</td>
<td>027C01C</td>
<td>00206</td>
<td>00207</td>
<td>1060</td>
<td>Increment record pointer</td>
</tr>
<tr>
<td>00027E</td>
<td>027E0404</td>
<td>00206</td>
<td>00207</td>
<td>1062</td>
<td>Blocked (must be fixed)</td>
</tr>
<tr>
<td>000280</td>
<td>0280180</td>
<td>00206</td>
<td>00207</td>
<td>1064</td>
<td>Copy number of bytes read</td>
</tr>
<tr>
<td>00028E</td>
<td>028E100</td>
<td>00206</td>
<td>00207</td>
<td>1066</td>
<td>Set lrec</td>
</tr>
<tr>
<td>000290</td>
<td>02900F0</td>
<td>00206</td>
<td>00207</td>
<td>1068</td>
<td>Emit the record</td>
</tr>
<tr>
<td>0002A0</td>
<td>02A007A</td>
<td>00206</td>
<td>00207</td>
<td>1070</td>
<td>Through buffer</td>
</tr>
<tr>
<td>0002B0</td>
<td>02B0180</td>
<td>00206</td>
<td>00207</td>
<td>1072</td>
<td>Slice off</td>
</tr>
<tr>
<td>00029A</td>
<td>029A040C</td>
<td>00206</td>
<td>00207</td>
<td>1074</td>
<td>Increment record pointer</td>
</tr>
<tr>
<td>00029B</td>
<td>029B056</td>
<td>00206</td>
<td>00207</td>
<td>1076</td>
<td>The lot</td>
</tr>
<tr>
<td>00029E</td>
<td>029E029</td>
<td>00206</td>
<td>00207</td>
<td>1078</td>
<td>No-one is listening...</td>
</tr>
<tr>
<td>0002C2</td>
<td>02C2015</td>
<td>00206</td>
<td>00207</td>
<td>1080</td>
<td>Yes, this is ok.</td>
</tr>
<tr>
<td>000000</td>
<td>0000137</td>
<td>00206</td>
<td>00207</td>
<td>1082</td>
<td>PROCEND RC=(R15)</td>
</tr>
</tbody>
</table>
Appendix B. Miscellaneous Reference Information

SYSPARM String

Assembler H (HASM) is used to assemble a module. The SYSPARM string is used to define which environment the module is to be assembled for, and how to handle test cases. No SYSPARM is required when there is one operand on FREETYPE, and when testing code is to be listed, but not generated.

SYSPARM is normally stacked by an exec which gets the information from a program directory; these options are only relevant if you are writing your own exec to assemble a module.

Two keywords with values can be coded:

- **SM** Defines the storage management desired. The following token should be one of OS, CMS, DYNAM. NONE indicates that the module must not allocate storage; procedure work areas must be pre-allocated, if needed.
- **TEST** The level of test code to generate. Three levels are supported:
  - **GEN** Generate code defined as test scaffolding.
  - **LIST** List the scaffolding but do not generate code for it. This is the default.
  - **EXCL** Exclude scaffolding; neither list nor generate this code.

The error listing (DDNAME SYSTERM) should be written to disk with file type ERRORS so it is available when editing to correct the errors.

Format of Program Directory Records

Program directories are stored in files with file type PGMDIR. A directory can be spanned over several minidisks; all occurrences are used for ASM PGMDIR; the first occurrence is used in the other ones.

Comment lines are identified by an asterisk in column one; such records are ignored. All other records must have a blank in the first column.

The Assembler Directory, ASM PGMDIR

The information in this directory is used when assembling a module.

- **2** Module name and storage management. This token has three components, the last two being optional.
  - The module name, storage management, and an optional qualifier separated by periods.
  - The file type in parentheses. The default is ASSEMBLE; other supported ones are PASCALVS, PLIOPT, and C.
  - Options to pass to the compiler. A leading period is optional.

- **29** The test code disposition. See “SYSPARM String” above.
The file name of the TXTLIB to receive the object module when it is assembled without updates.

Macro libraries to use during assembly. These are in addition to the library containing the macros defined in this book.

The Object Module Directory, TXT PGMDIR

The information in this directory is used to determine which load modules need to be regenerated when an object module has changed.

2 The TXTLIB name and the module name with a period in between.

20 A list of load module names to regenerate.

The Load Module Directory, MOD PGMDIR

This directory defines which object modules comprise a load module.

2 Load module name.

11 Load module type:

USER A CMS user area module.

XIENT A CMS user transient area module.

SYSXIENT A CMS system transient area module.

TEXT No module is generated; the composite TEXT deck is renamed to the load module name.

(Other) The token is the file name of a LOADLIB to receive the load module. The file is erased before the member is added to avoid the need to compress the library; this means that the library only has one member.

20 Object modules to include in the load module.

The list of object modules is a list of blank-delimited tokens each of which may be:

- The characters INCLUDE: followed by the file name of a file to include; the file type is INLCUDE. Lines of the file are chopped after column 72 (to allow for serial numbers) and joined together as a list of tokens. Lines are joined with a blank unless the last character of a line is a minus (-) which indicates that the following line is to be joined without a blank (and without the minus).

- The APL execute operator identifies a token to execute as a CMS command. The back-slash is used to indicate characters that should be blank when the command is executed. The command may be a single word with arguments in parentheses; The opening parenthesis is replaced with a blank and the closing one is discarded.

- The name of a TXTLIB with a list of members in parentheses. Members are separated by commas.

- The file name of an object module with file type TEXT.
Appendix C. Using Material Shipped with CMS/TSO Pipelines

This appendix describes how to generate a macro library with the macros described in this book from the CMS/TSO Pipelines product tape when it is intended to use the macros for other than CMS/TSO Pipelines maintenance.

Licensing Issues

The macro libraries contain restricted materials of IBM and may only be stored or used on processors that are licensed for CMS/TSO Pipelines, Program Offering 5785-RAC (in USA: RPQ P81059, 5799-DKF).

Once installed on a VM system, the macros can easily be shipped to an MVS system where they can be used just as well (since it is the same assembler); a separate licence is required if the MVS system runs on a processor with a serial number other than the one where CMS/TSO Pipelines is licensed.

Installation Procedure

The files you need are installed with the optional materials when CMS/TSO Pipelines is installed; you may be able to get them from there. Proceed with these tasks if the optional materials are not installed.

Allocate a Minidisk to Hold the COPY Files

Allocate 3 cylinders of 3380 or equivalent and define a minidisk for a suitable user. The access mode of the minidisk should be RR. The following discussion assumes the minidisk is on address 150.

Link the minidisk in write mode and format it with 4K blocks.

Load Files from the Product Tape

Inspect 5785RAC MEMO on the first file of the product tape to see if there is further information or alternate instructions. Proceed here if you find no instruction to the contrary.

Log on to the user that owns the minidisk defined above. Issue the commands:

```
link * 150 150 m
access 150 h
```

Ensure that no minidisks are accessed as a read-only extension of H.

These files will be required. On Modification Level 4 they are shipped in PIPUTIL PACKAGE on file 11 of the product tape. You can load the individual files listed here, but it may be easier to load the complete tape file.

PROC COPY: Procedures.
STROPTP COPY: Control flow.
GENINST COPY: Generalised instructions.
PRTASM EXEC: Listing post-processor.
PRTASMF MODULE: Filter package used by the listing post-processor.
Pipe CNTRL: A control file to apply updates.
PIPASM EXEC: Assembles a module; needs tailoring.

Unpack the COPY files; generate the macro library if there is no corrective service to apply.
copyfile * copy h (unpack olddate
maclib gen intlib proc strootp geninst
copyfile intlib maclib a = h2 (oldd repl
erase intlib maclib a

**Obtain Corrective Service**
Investigate if corrective service is available from the support centre; this tells you about *reported* problems.

A file with file type `AUXFIX`, and one or more update files constitute corrective service for a `COPY` file; load these files on disk H. Apply the update(s) and generate the macro library, for instance:

```
update proc copy h pipe cntrl h (ctl
maclib gen intlib $proc strootp geninst
copyfile intlib maclib a = h2 (oldd repl
erase intlib maclib a
erase $proc copy h
```

**Log off**
This releases the write link to the minidisk.

**Reporting Problems with the Macros**
The macros are supported to the extent of building an unmodified version of the program offering; by nature of building it, it is verified that they work: you cannot use formal IBM channels to report problems or obtain assistance with the macros. Send comments to the author at the address shown on the inside cover of this book. Though the author is keen to learn of problems, IBM Danmark A/S makes no commitment to fix any.
Glossary

Backward Reference. Reference in a PCALL to a procedure for which a PROC statement has been met at the time of reference.

Body. See procedure body.

Callee. When a procedure calls another one, the other procedure is the callee.

Caller. When a procedure calls another one, the first procedure is the callee.

Condition Code. IBM System/370 remembers the result of some instructions in the condition code which is tested by the branch on condition instruction. The condition code is a number from 0 to 3. The branch on condition instruction has a four-bit mask with a bit for each possible value; the instruction transfers control to the target when there is a one bit in the mask corresponding to the current condition code.

Containing Procedure. A procedure in whose work area a nested procedure has allocated its work area.

Dynamic. Allocated at execution time.

Enumerated Scalar. A variable that can take one of a listed set of values. It is declared with the macro SCALAR.

Epilogue. Code generated automatically to return from a procedure, releasing the work area if allocated in the prologue.

External Procedure. A procedure not in the module at hand.

Flag. A bit which can be tested for on (1) or off (0). A flag byte has up to eight flags; it is declared with FLAGDEF.

Forward Reference. Reference to an object that is not declared at that point in the assembly.

Forward Procedure Reference. (Forward procedure call.) Reference in a PCALL to a procedure that is declared later in the module.

Instance of a Variable. When a variable is declared in a procedure work area, storage is allocated for it when the procedure is entered. Procedures are often re-entrant, so multiple concurrent invocations are possible. A particular invocation of a procedure creates a particular instance of such a variable; each instance is separate from all other ones.

Internal Procedure. A procedure nested in the procedure at hand.

Invocation. A specific instance of a call to a procedure. A recursive procedure can have several invocations active at any time.

Level 1. The outermost level of procedure nesting. It is often more expensive in execution time to enter a level-1 procedure than a nested one.

Load Module. A core-image file containing a program. It often contains more than one object module.

Module. Usually means a source file.

MVS. Multiple Virtual Storages. An operating system for IBM System/370 computers.

Name Scoping. In high-level languages, variable names are often available only within the procedure where they are declared.

Nested Procedure. A procedure whose work area is allocated in a containing procedure’s work area. A nested procedure can only be called from a containing procedure.

Object Module. The relocatable TEXT deck created by the assembler.

OS. Operating System/360 and its derivatives: MVS, MVS/XA, and MVS/ESA.

Procedure. A piece of code that is called as a unit. It has one entry point and one exit. It can allocate a work area that is available while the procedure is executing.

Prologue. Code generated automatically at the entry to a procedure. The prologue saves general registers, establishes addressability, and allocates a work area if required.

Recursive. A procedure that calls itself is recursive. The recursion can be indirectly via one or more intermediary procedures.

Run time. When a program is run as opposed to when it is compiled.

Scalar. A single variable. See also enumerated scalar.

Scoping. See name scoping.

Stack. A data structure where elements are removed in the reverse order of their arrival. This is also called a last in first out (LIFO).

Stack Frame. A work area allocated on a stack by a procedure. High-level language implementations use a pointer into a large pre-allocated area to allocate and de-allocate stack frames as procedures are called and returns.
**Static.** Allocated at assembly time.

**Work Area.** Storage allocated for variables that are local to a procedure.
Index

Special Characters

: 62
- 62
: 63
? 62
/ 63
<

REGS 36, 44
###
Label Prefix 27
###$
Label Prefix 27
#A 47
#ALD 47
#C 47
#L 47
#LAL 49
#LT 47
#PUSEWA 37
#SLD 47
#ST 48
#SYNTAX 63
<DECREMENT>
  DECR 48
<ENTRY>
  MODEND 29, 58
<INCRENEMENT>
  INCR 48
<NAME>
  EXTPROC 30, 58
<number>
  62
<TARGET>
  INC 48
= 63

Numerics

0
MIN 67
0TO15 63

1
LOCTR 37
16M line 55
5785RAC
MEMO 77
8
MIN 58

A
ABOVE
  PIPDESC 55
Addressability errors 64

ALL 25
ALOC 44
AMODE
  MODBEG 28
ANY 28
APL execute 76
ARGS
  PIPDESC 55
Argument structure 50
AS
  XEDIT 24
ASCM
  XEDIT 24
ASM
  HLL 50, 51, 53, 54, 55
  PGMDIR 75
ASMM
  EXCE 22
ASSEMBLE
  Filetype 17, 18, 21, 22, 23, 25, 75
ASSERT 39
ATYPE
  PIPEPTEN 58
AUTOEXT 29, 35
AUTOSIZE
  YES 44
AUXFIX
  Filetype 78
  Filetype 22

B
B 40
Backward Reference
  Glossary definition 79
BAL 4, 28
BALR 4, 28, 36
BAS 28, 32, 35, 66
BASE 43, 69
BASR 9, 28, 35, 36
BCR 35
BCT 13, 42
BCTR 48, 65
Body
  Glossary definition 79
BORROW 38
BUFFER 56
  PIPDESC 55
BUSY 38
BXH 13
BXL 13, 42
BYTES 43
C

CALL 63
Callee
Glossary definition 79
Caller
Glossary definition 79
CALLSYNT 63
CARRY 38
CASBEG 39
CASITEM 39, 40
CC 36, 71
PEXIT 36
RETAIN 36
CHACT
XEDIT 23
CHAIN 42
CHANNELBUSY 38
CLEAR 43
YES 43, 69
CLI 12, 40
CMS 75
MODBEG 28
CMSSYS
MODBEG 28
CNTRL
MODBEG 28
CNTRL
Filetype 18
PIPE 77
COMMIT
PIPEDESC 56, 58
COMPLETE 38
COND 35, 39, 41
Condition Code
Glossary definition 79
Conditions Codes 38
CONDSUP 44
CONT
XEDIT 24
Containing Procedure
Glossary definition 79
CONTINUE 42
COPY 27
Filetype 18, 21, 22, 23, 25, 77
GENINST ii, 77
PIPEIF ii
PROC ii, 69, 70, 77
STROPTP ii, 77
Corrective service 78
CP
MODBEG 28
CSECT 28, 47, 49
CSWSTORED 38
EXTPROC  30
  <NAME>  30
EXTRN  30

F
  FAR
    YES  4, 35
  FI  39

Filetypes
  ASSEMBLE  17, 18, 21, 22, 23, 25, 75
  AUXFIX  22, 78
  C  75
  CNTRL  18
  COPY  18, 21, 22, 23, 25, 77
  ERRORS  21, 75
  INLCUDE  76
  LINTERRS  18, 23
  LINTOK  18, 23
  LOADLIB  76
  PASCALVS  75
  PGMDIR  17, 75
  PLIOPT  75
  TEXT  17, 76, 79
  TXT  18
  TXTLIB  17, 18, 76

Filter package  51, 57
  FIRST  47
  PIPDESC  55

Flag
  Glossary definition  79
  FLAGDEF  45
    INIT  45
  FLAGEQU  45
  FLAGOFF  46
  FLAGON  45
  FLAGS
    PIPDESC  56
  FLAGSET  46
  FLAGTM  45
  FLAGXI  46

Forward Procedure Reference
  Glossary definition  79

Forward Reference
  Glossary definition  79

FP
  PIPDESC  55
  FPGXCOM
    GBL  53
FREE  43
FREEMAIN  28
FREETYPE  69, 72, 75
  MODBEG  28
  NONE  51, 68, 69
FRET  43
FWDONLY
  PROC  33

FWDONLY (continued)
    YES  70
FWDPROC  31

G
  GBL
    PIPDESC  56
    FPGXCOM  53
  GEN  29
    PRINT  29
  NO  35, 67
  GENEND
    MODEND  29

General register
  0  18, 34, 43, 44, 61, 63
  0 through 11  33
  1 3, 9, 10, 30, 33, 34, 35, 36, 43, 44, 62, 63, 66
  12 3, 18, 33
  13 3, 9, 30, 33
  14 3, 33, 43
  15 3, 33, 35, 36, 43, 47, 52, 53, 60, 61, 62, 63, 69
  2 52, 54
  3 52, 54
  3 through 11  33
  4 55, 61
  5 51, 53, 54, 61, 63
  6 53
  9 51
  GENINST
    COPY ii, 77
  GENOFS  49
  GETMAIN  28
  GOTOEXIT  35

H
  HIDE  9, 33
  PROC  33
  HIGH  38
  HLL  50
    PIPDESC  55
    ASM  50, 51, 53, 54, 55
  HLLSTACK  31, 33
    SAREA  6, 9, 28, 29, 30, 33, 34, 52

I
  1
    KNOW  42, 72
  ICM  13, 42
  ID  38, 67
  IF  13, 38
  INCOMPLETE  38
  INCR  42, 48
    <TARGET>  48
ONE  38
ONES  38
Operations for syntax machine  62
OPSYN  5, 28, 34
OS  75
   MODBEG  28
   Glossary definition  79
OVERFLOW  38

P
PAGEINVALID  38
PARM  35
PASCALVS
   Filetype  75
PBEGIN  34
   EP  34
PCALL  7, 35
PDBGCODE  31
PEXIT  36
   REGS  36
PGMDIR
   ASM  75
   Filetype  17, 75
PGMLANG
   PIPEPTEN  58
PIPEASM
   EXEC  77
PIPDESC  54
   ENTRY  55, 56
PIPE
   CNTRL  77
PIPEIF
   COPY ii
Pipeline services transfer vector  51
PIPLCATE  57
PIPEPTED  58
PIPEPTEN  57
   <NAME>  58
PILOPT
   Filetype  75
PLTORG  37
PLUS  38
POSITIVE  38
PRINT  29
   MODBEG  28, 29
   GEN  29
PROC  32
   ENTRY  32, 33
   COPY ii, 69, 70, 77
Procedure
   Glossary definition  79
PROCEND  37
PROFILE
   XEDIT  23

Program descriptor  50, 50
Prologue
   Glossary definition  79
PRTASM
   EXEC  77
PRTASMF
   MODULE  77

Q
Q
   XEDIT  23

R
R1  63
RC  36
PEXIT  36
Recursive
   Glossary definition  79
REG  40, 68
   INCR  48
REGS  36
   PEXIT  36
   (  36
REPEAT  41
RETAIN
   CC  36
RETURN  29, 62
REXSTR  21, 26
   XEDIT  25
RMODE
   MODBEG  28
Run time
   Glossary definition  79

S
S  48
S-type constant  32
SAREA  9, 28, 71
   PROC  33
   STACK  3, 5, 6, 9, 10, 18, 28, 29, 30, 33, 34, 35, 36, 52,
       66, 69, 71
SAVE
   PROC  32
   NO  5, 30, 33, 35, 36, 66, 69, 71
SCALAR  47
   Glossary definition  79
Scoping
   Glossary definition  79
SEGMENTINVALID  38
SELFRELOCATE
   MODBEG  28
SETC  34, 39, 40
SETLENGTH  43
   YES  43, 44
CMS/TSO Pipelines: Procedures Macro Language