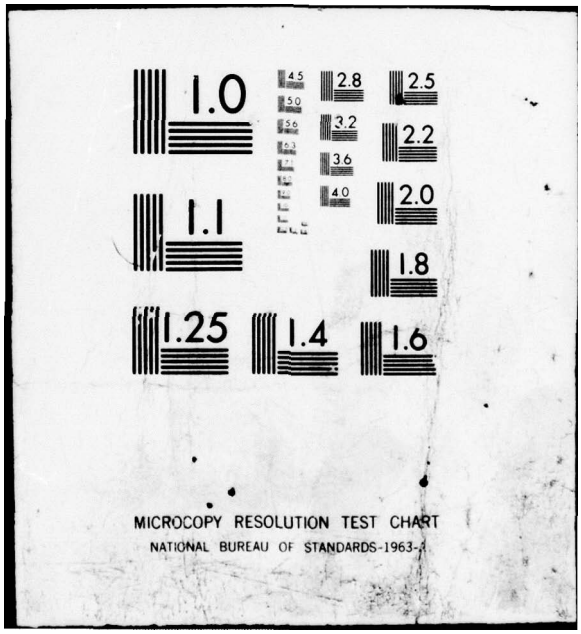
AD-A074 496 SRI INTERNATIONAL MENLO PARK CA F/G 9/2
ON-LINE PROGRAMMER'S MANAGEMENT SYSTEM. ADDENDUM II. PROGRAMMER--ETC(U)
AUG 79 B L PARSLEY, H G LEHTMAN, S KAHN F30602-77-C-0185
UNCLASSIFIED RADC-TR-79-205-ADD-2 NL

1 OF 3
AD
AD74496




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD A074496

RADC-TR-79-205, Addendum II
Final Technical Report
August 1979

LEVEL II
A 074 495

ON-LINE PROGRAMMER'S MANAGEMENT SYSTEM
Programmer's Guide to the Debugger

Augmentation Resources Center

Bruce L. Parsley
Harvey G. Lehtman
Susan Kahn

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

DDC
RECEIVED
OCT 1 1979
A

DDC FILE COPY

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

79 10 01 045

This report contains a large percentage of machine-produced copy which is not of the highest printing quality but because of economical consideration, it was determined in the best interest of the government that they be used in this publication.

This report has been reviewed by the RADC Information Office (OI) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-79-205, Addendum II has been reviewed and is approved for publication.

APPROVED: Raymond A. Liuzzi

RAYMOND A. LIUZZI
Project Engineer

APPROVED: Wendall C. Bauman

WENDALL C. BAUMAN, Colonel, USAF
Chief, Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISIE), Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DDC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: RADC-TR-79-205, Addendum II (ADD-2)	2. GOVT ACCESSION NO. (Addendum II.) / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): ON-LINE PROGRAMMER'S MANAGEMENT SYSTEM. Programmer's Guide to the Debugger.	5. TYPE OF REPORT & PERIOD COVERED: Final Technical Report. Sep 77 — Mar 79
	6. PERFORMING ORG. REPORT NUMBER: N/A
7. AUTHOR(s): Bruce L. Parsley, Harvey G. Lehtman, Susan Kahn	8. CONTRACT OR GRANT NUMBER(s): F30602-77-C-0185 (SRI Menlo Park CA)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Augmentation Resources Center, 20705 Valley Green Drive, Cupertino CA 95014	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 55811803 (17) 18
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (ISIE), Griffiss AFB NY 13441	12. REPORT DATE: Aug 1979
	13. NUMBER OF PAGES: 216
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Same (12) 215p.	15. SECURITY CLASS (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Raymond A. Liuzzi (ISIE)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Debugging	System Software
On-Line	JOVIAL
Software Engineering	Compilers
Programming Environments	
Computers	

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report is composed of studies that have been conducted to develop the NLS system as an on-line programming environment and to provide an on-line JOVIAL interactive debugger with the capabilities to debug JOVIAL language programs. The final report contains several design additions to the NLS system to create an on-line programming environment. A JOVIAL User's Guide prepared in Addendum Technical Report I provides an extensive set of commands for using the JDAD Debugger. Addendum Technical Report II provides a generalized approach to debugging and describes the NLS/NSW Do-All Debugger (DAD).

DD FORM 1 JAN 73 1473

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

410 281

INTRODUCTION 1

Through the years there have been a number of approaches to, and aids for, the debugging of programs. The approaches have varied from sitting down at a CPU console and toggling switches and reading lights, to highly sophisticated, high level language interactive debuggers. 1a

This debugger attempts to be a fairly sophisticated high level language interactive extensible debugger. It is designed and organized so that as new techniques and/or languages and/or machines become available, they can be easily integrated with the rest of the debugger in a coherent manner. 1b

The debugger contains very few, if any, techniques that have not been used in other debuggers. What is unique about this debugger is its internal organization that makes it possible to be almost all things to almost everyone. (More on this grandiose statement as we go on.) 1c

Accession For
NTIS GRA&I
DDC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or special
A 23

WHAT IS DEBUGGING AND THE ORGANIZATION OF THIS DEBUGGER 1

When one thinks about the process of interactive debugging, the tasks that must be performed fall roughly into the following categories: 2a

the user (the programmer who is doing the debugging) must be able to specify an action that he wishes performed;

this user functional specification must be translated into a call on specific debugger routines;

these debugger routines must be able to read and write the bits of the program being debugged, and be able to read and modify the state of the program being debugged;

these bits from the program, and the program's state information, must be interpreted in a manner that will be meaningful to the user, and in a manner consistent with the language the program is written in;

and finally, the resulting interpretations, must be presented to the user.

This debugger recognizes these functions and is organized regularly, with separate modules for each of these functions. 2b

Thus, there exists a debugger frontend module for all communication with the user. This module consists of, among other parts, a Command Language Interpreter, a grammar that represents the commands of the debugger, and communication code for communicating user specified commands to the rest of the debugger and for receiving information from the rest of the debugger for display to the user.

There exists a debugger dispatcher (DD) module that receives functional command specifications from the frontend and calls various routines to implement these requests and transmit user meaningful results back to the frontend.

There exists an operating system module (OSM) with responsibility for reading and writing the bits and state information of the program being debugged.

There exists a language module (LM) for interpreting the bits and state information of the program being debugged in a manner appropriate to the language that the program was written in and

appropriate to the machine and operating system on which the program is running.

With the use of this modular approach, along with well defined and published functional specifications for each module and for inter-module communication, the following benefits become feasible: 2c

as long as each module conforms to its functional specifications and communication standards, it can implement its functions in any manner, and in any language, it chooses;

it is possible to dynamically plug in and out individual modules.

It is this approach that enables the debugger to be extensible and to "be all things to everybody". Consider the following: 2c

a frontend module, F, that implements a specific command language discipline;

a debugger dispatcher module, D;

an OSM, YOSM, for the reading and writing of bits and state information of programs written for machine and operating system Y;

a LM, XLM, for the interpreting of bits and state information of programs written in language X; and

a program, PX, written in language X to run on machine and operating system Y.

The 4 modules, F, D, YOSM, and XLM, provide all the ingredients needed to debug program PX. Now if we have a program, PN, written in language N to run on machine and operating system Y, and we have a LM, NLM, for the interpreting of bits and state information of programs written in language N, then all we have to do to debug program PN, is to replace the XLM with the NLM. This same process, of replacing modules, enables us to debug not only programs written in different languages on the same machine, but by replacing the OSM, we can also debug programs running on different machines and operating systems. 2e

SOME INITIAL TERMINOLOGY 3

The following terms will be used throughout this document (hopefully in a consistent manner and with their generally accepted meanings): 3a

a user is a human being who wishes to do some debugging;

the user will interact with the debugger in real time through an interactive terminal (as opposed to using cards for example);

a program is a sequence of computer instructions;

the term routine will be used to refer to either a procedure or a coroutine;

a process is an instance of a program that is running on some computer and thus has state information associated with it, and has one, and only one program counter (PC) associated with it;

a process also has associated with it an address space; this address space refers to the logical addresses available to the process and does not refer to physical memory;

the term target process (or current target process) will refer to the process which is currently being debugged by the debugger.

the terms debugger frontend or frontend process or simply frontend refer to that process (or that part of the debugger) which is responsible for interacting with a user.

the term frontend machine refers to the logical machine on which the debugger frontend process is currently executing.

the terms debugger backend or backend process or simply backend refer to that process which is composed of the DD, the LM, and the CSM.

COMMUNICATION WITHIN THE DEBUGGER 4

Several communication paths exist in the debugging environment we are talking about. There is a communication path between a calling process, normally the debugger frontend, and the debugger backend (or more specifically the debugger dispatcher; this will be referred to as frontend-backend communication); there are communication paths among the 3 modules comprising the backend (which will be referred to as inter-module communication); there are communication paths between routines within one module (intra-module communication); and there is a communication path between the debugger backend and the target process (referred to as backend-target communication). 4a

FRONTEND-BACKEND COMMUNICATION 4b

Communication over this path will conform to SAFE (Stand Alone FrontEnd) communication protocol standards. All communication from the frontend to the backend is actually received by one routine in the debugger dispatcher. This routine will convert all "messages" received into internal debugger format (discussed below) and then call the appropriate routines. Similarly, all communication from the backend to the frontend goes through a small number of routines that convert from internal debugger format to NSW communication protocol standards.

INTER-MODULE COMMUNICATION 4c

All inter-module communication must conform to internal debugger format. This is discussed below.

INTRA-MODULE COMMUNICATION 4c

Intra-module communication standards are the responsibility of the individual modules. One of the benefits of this modular approach to the debugger is that the actual details of the implementation of individual modules is of no concern to the other modules. Thus, a module can adopt whatever standards for itself that it wants as long as it meets its functional specs and conforms, where necessary, to communication standards for communicating with other modules.

Intra-module communication for the debugger dispatcher module, the L10 language module, the JOVIAL language module, and the TENEX operating system module, conform to internal debugger format.

BACKEND-TARGET COMMUNICATION 4e

In order to debug any process in an interactive manner, there is a small set of primitive functions that must be provided either by the target process, or by a process that has control over the target process. These functions include such things as reading or writing the address space of the target process, etc.

(See the appendix for a discussion of these primitives.)

The CSM (and only the OSM) is responsible for debugger communication with the target process. It is expected that as well as having different OSMs for the debugging of processes on different machines, there may also be different protocols for communication between OSMs and target processes.

GENERAL COMMENTS ON MODULES 5

Since the computer field has not yet achieved true machine independence of programming languages (to the author's knowledge anyway), each module is written to run in a specific environment that includes the machine and the operating system the module will be executing under. Thus, there is a DD module written to run in a TENEX environment, and a different, but functionally identical, DD module to run under an OS/360 environment. Also, there is a L10 LM to run under TENEX, and a different L10 LM to be run under an OS/360 environment. 5a

It is strongly recommended, and existing debugger modules follow this recommendation, that modules be organized along the following lines: 5b

Modules should be written in a high level language, and preferably one that is available to more than one machine and operating system environment.

Each module should consist of the following 2 major parts:

A part that is independent of the machine and operating system that the module is to run on. This would include any code written in the high level language that would run regardless of what environment the module were running under. For example, an arithmetic statement in any language is likely to compile and run successfully in any environment that supports the specific high level language.

A part that is dependent on the machine and operating system that the module is to run on. This includes any code that would have to be recoded to run in a different environment. For example, any I/O code would most likely have to be rewritten if the module were to be run in a different environment.

The purpose of these recommendations is to try to minimize the amount of work needed to transport the debugger from one environment to another. In addition, it has been found that code organized along these lines is easier to maintain and modify. 5c

THE LANGUAGE OF THE DEBUGGER AND DEBUGGER FORMATS 6

THE FRONTEND 6a

The debugger grammar is specified in CML. Parsefunctions and selection routines, needed to augment the grammar, are written in L10, and conform to the standards for parsefunctions and selection routines.

THE BACKEND 6b

The debugger dispatcher (to be run under TENEX), the L10 language module (to be run under TENEX), the JOVIAL language module (to be run under TENEX), and the TENEX operating system module (also to run under TENEX), are currently written in L10.

The internal debugger format currently conforms to L10 conventions for:

the invocation of routines,

the passing of arguments and the returning of results,

and the referencing of data structures across modules.

This does not mean, however, that a LM or an OSM cannot be written in a language other than L10. A LM or an OSM can be written in any language as long as all inter-module communication, all data structures that will be referenced by other modules, and all referencing of data structures in other modules, conform to the debugger formats, i.e. L10 conventions.

DISPATCH TABLES 7

Since each module in the debugger is designed to be relatively self-certained and to be dynamically loaded and unloaded as needed, a mechanism is required for one module to obtain the address of a routine or data structure in another module that it needs to use. The mechanism adopted is that each module provides a fixed formated data structure, called the dispatch table, that lives in a fixed place. Entries in this data structure are then the addresses of routines and/or data structures provided by the module (and in some cases an entry in the dispatch table may be a data structure). The meaning of a specific entry in a dispatch table is predefined. For example, the first entry in the dispatch table for any LM is the address of the LM's initialization routine. 7a

A routine whose address is contained in a dispatch table, i.e. a routine that may be called from a module other than the one in which it physically exists, will be called an external routine. Similarly, a data structure whose address is contained in a dispatch table (or which exists as an entry or entries in a dispatch table) will be called an external data structure. 7b

DATA STRUCTURES 8

There is a need for data structures that can be maintained and/or referenced by more than one module. Data structures of this type will be referred to as external data structures. In some simple cases, a dispatch table entry will itself be a data structure and such data structures will be called simple data structures. A module references an external data structure by finding out the address of the data structure from the appropriate dispatch table, or in the case of simple data structures by referencing the dispatch table. 8a

CHARACTER SETS

Since the debugger is designed to support a number of different languages, and since most languages do not use the same character sets as valid characters in identifiers, etc., the interpretation of strings input by a user cannot be handled by the DD but must be handled by individual LMs. In order to maintain some consistency for users while debugging many languages, the following approach has been adopted:

The DD contains an external data structure, called the Generic Function String (GFS), that is a 128 character L10 string whose address is in the DD dispatch table. The ith character of this string contains a value that represents the generic function for ascii character code i.

When a LM wishes to interpret user input strings, the LM must look up each character in the user input string in the DD GFS to determine the function of the user character and then act accordingly.

If a LM wishes to modify the GFS it MUST use the DD external routine whose address is at offset 0D0H in the DD's dispatch table.

For documentation and communication purposes, it is convenient to have a generic name to refer to the character that is currently serving a specific generic function. Thus, while the specific character may change, it can still be referred to by its generic name. The generic name for a character is the uppercase word of the generic function symbolic name, e.g. the generic name for the character that is currently serving the generic function of an access list delimeter (semicolonchar) is SEMICOLONCHAR.

The symbolic names for the generic function values are maintained in the file <rsw-debugger>dotdef.rls and are defined in the debugger loader. Thus, these symbolic values are available for all modules. The symbolic names and the meaning of these generic functions are as follows (the debugger default character, in the absence of user or LM modification, for a generic function will appear under the meaning column delimited by a left angle bracket (<) and a right angle bracket followed by a semicolon (>;)):

generic function symbolic name	meaning of character i
normchar	this character is a normal character that can be interpreted in any manner the LM chooses
pluschar	<+>; the user is using this character as the arithmetic addition operator
minuschar	<->; the user is using this character as the arithmetic subtraction operator
timeschar	<*>; the user is using this character as the arithmetic multiplication operator
divicechar	</>; the user is using this character as the arithmetic division operator
lparenchar	<(>; the user is using this character as the arithmetic left grouping character
rparenchar	<)>; the user is using this character as the arithmetic right grouping character
blockchar	<%>; the user is using this character as a block delimeter; e.g. the string: string1%string2 should be interpreted as symbol string2 in block string1 if % is the current BLOCKCHAR
fieldchar	<.>; the user is using this character to delimit the fields of a record: if the current language does not support records, then this character may be interpreted as a normchar
escapechar	<altmode or escape>; the user is using this character to mean interpret the next character as a debugger builtin variable; e.g., ESCAPECHAR followed by a 'C (or 'c) refers to the builtin debugger variable which has the value of the last displayed cell

generic function symbolic name	meaning of character i
spacechar	<space>; the user is using this character as a space character; the SPACECHAR should normally be interpreted as an entity delimeter, however, in the evaluating of address range elements, two strings separated only by SPACECHARs should be interpreted as having a PLUSCHAR between the 2 strings
commachar	<,>; the user is using this character as an address range delimeter to separate the two elements of an address range; under normal circumstances, a LM will never see the COMMACHAR in user input strings
semicolonchar	<;>; the user is using this character to separate address ranges within address lists; under normal circumstances, a LM will never see the SEMICOLONCHAR in user input strings
larrowchar	<_>; the user is using this character as the debugger assignment character; under normal circumstances, a LM will never see the LARROWCHAR in user input strings
tabchar	<tab>; the user is using this character to mean display the cell accessed by the most recently displayed cell; under normal circumstances, a LM will never see the TABCHAR in user input strings
poundchar	<#>; the user is using this character to mean back up to the previous displayed cell; under normal circumstances, a LM will never see the POUNDCHAR in user input strings
lfchar	<linefeed>; the user is using this character to mean display the next sequential cell; under normal circumstances, a LM will never see the LFCHAR in user input strings

generic function symbolic name	meaning of character i
uparrowchar	<|>; the user is using this character to mean display the previous sequential cell; under normal circumstances, a LM will never see the UPARROWCHAR in user input strings
bslashchar	<\>; the user is using this character to mean display an address list in string mode; under normal circumstances, a LM will never see the BSLASHCHAR in user input strings
equalchar	<=>; the user is using this character to mean display the value of the input address list; under normal circumstances, a LM will never see the EQUALCHAR in user input strings
excmarkchar	<!>; the user is using this character to mean display cells as ascii values; under normal circumstances, a LM will never see the EXCMARKCHAR in user input strings
lsquarechar	<[>; the user is using this character to mean display an address list numerically; under normal circumstances, a LM will never see the LSQUARECHAR in user input strings
qmarkchar	<?>; the user is using this character to mean tell where symbols in an address list are defined; under normal circumstances, a LM will never see the QMARKCHAR in user input strings
rsquarechar	<]>; the user is using this character to mean display an address list as records; under normal circumstances, a LM will never see the RSQUARECHAR in user input strings
slashchar	</>; the user is using this character to mean display an address list symbolically; under normal circumstances, a LM will never see the SLASHCHAR in user input strings

ADDRESS LISTS 10

DISCUSSION 10a

An address list is the basic manner in which a user refers to elements in the target process. Basically, an address list is composed of one or more address ranges; and an address range consists of one or two address range elements (AREs).

(The character that terminates an address list, while it may modify the functional use of the address list, is not a part of the address list itself.)

Due to the wide variety of syntactical and semantical rules of different languages for expression evaluation, it is not possible for the DD to evaluate address ranges. Thus many LM external routines take as input the addresses of the 2 ARE strings that compose an address range and it is the responsibility of the LM to evaluate the AREs.

However, each LM is expected to obey certain debugger standards in the evaluation of the AREs:

all characters must be checked to determine what generic function they are currently serving (see the above discussion on character sets),

before evaluating an individual ARE, the LM must call the DD external routine whose address is at offset dddddd in the DD's dispatch table to determine the gross type of the address range it has received.

The following are the symbolic names (available to LMs since the definitions are a part of the debugger loader) and the meanings of the gross address range types:

dill -

this is an illegal address range

dmem -

this address range refers to cells in the address space of the target process

cfra -

this address range refers to stack frames that reflect the target process' current language state for stack oriented languages

cfcr -

this address range refers to the formal parameters of a procedure in a procedural oriented language

dcat -

this address range refers to the catchphrases for a procedure in a procedural oriented language that supports exceptional clauses (CATCHPHRASEs in L10; ON CONDITIONs in PL1)

dsig -

this address range refers to the signal status of the target process

dacr -

this address range refers to the memory utilization of the address space of the target process (e.g. a TENEX EXEC MEMSTAT command)

if a specific area type is not supported by a specific LM, it must generate the appropriate error return rather than interpreting the ARE in some other manner.

ADDRESS LIST TERMINATORS 10b

The user may terminate an address list with a number of different characters, depending on which command he is specifying. The terminating character is NOT a part of the address list, and is passed to the DD, by the CLI, as a separate argument. The stripping off of the terminating character is handled by the selection routines. The following are the generic characters, with their meaning, that may be used to terminate various address lists:

generic character terminator	meaning
LARROWCHAR	after each line of the address list is displayed, the user wishes to assign a new value to the just displayed entity
BSLASHCHAR	the user wishes to see the address list displayed in string mode
EQUALCHAR	the user wishes to have the value of the input address list displayed to him
EXCMARKCHAR	the user wishes to see the address list displayed in ascii mode
LSQUARECHAR	the user wishes to see the address list displayed in numeric mode
QMARKCHAR	the user wishes to find out where the symbols in the entered address list are defined
RSQU/RECHAR	the user wishes to see the address list displayed in record mode
SLASHCHAR	the user wishes to see the address list displayed in symbolic mode

FORMAL DEFINITION 10c

```
ADRLIST := ADRRANGE [ SEMICOLONCHAR ADRLIST ]
ADRRANGE := RANGE / BUILTIN / RECORDSPEC
BUILTIN := FRAME / PARAM / SIGNAL / CATCH / MEM / PLIST
PLIST := ESCAPECHAR ('Z / 'z)
MEM := ESCAPECHAR ('M / 'm)
CATCH := ESCAPECHAR ('C / 'c)
SIGNAL := ESCAPECHAR ('S / 's)
PARAM := ESCAPECHAR ('P / 'p)
FRAME := FSPEC [ COMMACHAR FSPEC ]
FSPEC := FF / FR / FO / FT / FB
FF := ESCAPECHAR ('F / 'f)
FR := ESCAPECHAR ('F / 'f) ('+ / '-) [ NUMBER ]
FO := ESCAPECHAR ('F / 'f) ('O / 'o)
FT := ESCAPECHAR ('F / 'f) ('T / 't)
FB := ESCAPECHAR ('F / 'f) ('B / 'b)
RECORDSPEC := EXPRESSION FIELDCHAR EXPRESSION
RANGE := EXPRESSION [ COMMACHAR EXPRESSION ]
EXPRESSION := TERM [ OPERATOR EXPRESSION ]
TERM := IDENT / LPARENCHAR TERM RPARENCHAR
OPERATOR := PLUSCHAR / MINUSCHAR / TIMESCHAR / DIVIDECHAR /
SPACECHAR
IDENT := BLOCKIDNT / SMPLIDNT / NUMBER / BLTNTRM
BLOCKIDNT := SMPLIDNT BLOCKCHAR SMPLIDNT
SMPLIDNT :=
   a string composed of valid identifier characters for the
   current language
BLTNTRM := BL / BO / BA
BL := ESCAPECHAR ('L / 'l)
BO := ESCAPECHAR ('O / 'o)
BA := ESCAPECHAR ('A / 'a)
NUMBER := a string of digits in the current input mode radix
SEMICOLONCHAR :=
   the character currently serving the generic function of
   semicolonchar
COMMACHAR :=
   the character currently serving the generic function of
   commachar
FIELDCHAR :=
   the character currently serving the generic function of
   fieldchar
ESCPAECHAR :=
   the character currently serving the generic function of
   escpaechar
```

SPACECHAR :=
the character currently serving the generic function of spacechar
LPARENCHAR :=
the character currently serving the generic function of lparenchar
RPARENCHAR :=
the character currently serving the generic function of rparenchar
PLUSCHAR :=
the character currently serving the generic function of pluschar
MINUSCHAR :=
the character currently serving the generic function of minuschar
TIMESCHAR :=
the character currently serving the generic function of timeschar
DIVIDECHAR :=
the character currently serving the generic function of divicechar

SEMANTICS 10c

SPACECHARs should be ignored except in the evaluation of an EXPRESSION or after the '+ / '- in an FR

one or more SPACECHARs separating TERMs of an EXPRESSION will be used to mean addition unless the SPACECHARs are adjacent to an arithmetic operator, to the right of a LPARENCHAR, or to the left of a RPARENCHAR.

EXPRESSIONs should be evaluated in a left to right, non-hierarchical order. However, evaluation can be modified by the use of LPARENCHARs and RPARENCHARs.

PLIST := ESCAPECHAR ('Z / 'z)

used as a shorthand notation to be equivalent to the previously typed in access list

MEM := ESCAPECHAR ('M / 'm)

used to show the utilization of the address space of the target process

CATCH := ESCAPECHAR ('C / 'c)

Used to show the catchphrases for the current frame.

SIGNAL := ESCAPECHAR ('S / 's)

Used to show the signal status of the process.

PARAM := ESCAPECHAR ('P / 'p)

Used to show the formal parameters of the current frame

FF := ESCAPECHAR ('F / 'f)

FF refers to the current frame. the current frame is the most recently displayed frame or the frame on the top of the stack after the debugger context is established (via hitting a breakpoint or starting to debug a target process).

FO := ESCAPECHAR ('F / 'f) ('O / 'o)

Used to show the owner frame of the current frame; the owner of a procedure is its caller; the owner of a coroutine is the routine that did the operport to the coroutine.

FT := ESCAPECHAR ('F / 'f) ('T / 't)

Used to show the top frame on the stack

FB := ESCAPECHAR ('F / 'f) ('B / 'b)

Used to show the bottom frame on the stack

FR := ESCAPECHAR ('F / 'f) ('+ / '-) [NUMBER]

If NUMBER is not specified it defaults to 1: no SPACECHARs may precede NUMBER; NUMBER specifies the number of frames to move relative to the current frame; e.g. if 's is the current ESCAPECHAR, and ', is the current COMMACHAR, the FRAME: "sft, sf-2" would display the frame on the top of the stack, and the next two frames towards the bottom of the stack in the control thread.

RECORDSPEC := EXPRESSION FIELDCHAR EXPRESSION

RECORDSPEC is used to represent a field (specified by the second EXPRESSION) of the record instance at the address specified by the first EXPRESSION; e.g. if period is the current FIELDCHAR, then the RECORDSPEC: "rec.fld" refers to field "fld" of the record instance at address "rec".

BLOCKIDNT := SMPLIDNT BLOCKCHAR SMPLIDNT

BLOCKIDNT is used to refer to the (local) symbol (specified by the second SMPLIDNT) in the block (or file) specified by the first SMPLIDNT; e.g. if '$ is the current BLOCKCHAR, then the BLOCKIDNT: "fl$sfilev" would refer to the symbol "sfilev" in file "fl".

BL := ESCAPECHAR ('L / 'l)

this entity has the value of the most recently completely evaluated EXPRESSION

BC := ESCAPECHAR ('C / 'c)

this entity has the value of the most recently displayed cell

BA := ESCAPECHAR ('A / 'a)

this entity has the value of the address of the most recently displayed cell

DEBUGGER WIDE DATA STRUCTURES 11

Many external routines in the language module must maintain certain data structures in the DD module. They do this either by calling routines in the dispatcher (through the DD's dispatch table) or by manipulating the data structures directly (once again, however, the location of the data structure is obtained through the DD's dispatch table). It is the responsibility of language module external routines to see that the following data structures are kept current: 11a

LSTVDIS - this is a simple data structure which consists of one cell in the DD dispatch table which contains the most recently displayed value

(the user represents this value by entering ESCAPECHAR-D)

LSTEADR - this is a simple data structure which consists of one cell in the DD dispatch table which contains the value of the most recently evaluated access range element

(the user represents this value by entering ESCAPECHAR-L)

LSTADIS - this is a data structure containing the addresses of the last n displayed cells

(n is currently set to 4)

(language module routines maintain this data structure by using the DD external routine whose access is at offset octal 6 in the DD's dispatch table)

(the user represents this value by entering ESCAPECHAR-A)

INPUT / OUTPUT MODE RECORDS 12

Many external routines take as arguments the address of the current input or output mode records. These records lie at the heart of the debugger and are used to govern the way input from a user is interpreted and the way output is formatted. 12a

The DD actually maintains a permanent input mode record and a current input mode record, and a permanent and a current output mode record. Before calling any UM or DSM routines, the DD will set up the current input and output mode records based on their respective permanent values and any current modifiers specified by the user for the current command.

There exist user commands to examine and modify the permanent values of these records.

Both the input and output mode records are debugger formatted records, and what follows is the PL/I declarations for these records and an explanation of the possible values and meaning of the individual fields. 12b

(All the following symbolic definitions are a part of the debugger loader and are thus available to all UMs.)

THE INPUT MODE RECORD 12c

```
(inmode) RECORD
   ihlang[5],
   iclang[2],
   iradix[5],
   itrace[5],
   ibytesize[6],
   inname[ADDRESS];
```

field	possible values	meaning
ihlang		this field specifies what the current high level language for input is; i.e. if the current level of language in use (as specified by 'iclang') is highlanguage, then user input should be considered to conform to the semantical and syntactical rules of this high level language

field	possible values	meaning
ihlang	l10	the current high level language in use is L10
	cobol	the current high level language in use is COBOL
	fortran	the current high level language in use is FORTRAN
	bcpl	the current high level language in use is BCPL
	pl1	the current high level language in use is PL1
	jovial	the current high level language in use is JOVIAL
iclang	this field specifies what level of language should be used for the interpretation of user input	
iclang	machine	the current level of language in use is machine language
	assembly	the current level of language in use is assembly language
	highlanguage	the current level of language in use is the current high level language
iradix	a number	all numeric input should be interpreted as being numbers in the base specified by this field
itmode	this field specifies what the current input mode is	
itmode	tmcurlang	user input should be interpreted according to the current language specifications of fields iclang and ihlang

field	possible values	meaning
	tmascii	user input should be interpreted as ascii values
	tmsixbit	user input should be interpreted as sixbit values
	tmrad50	user input should be interpreted as radix 50 values
	tmfloat	user input should be interpreted as floating point numbers
	tmbyte	user input should be interpreted as successive bytes, with each byte having a bytesize as specified by ibytesize
ibytesize	a number	bytesize to use if the current input mode is tmbyte
irname	---	currently unused

THE OUTPUT MODE RECORD

120

```
(outmode) RECORD
   chlang[3],
   oclang[2],
   cradix[5],
   ctmode[5],
   ctypesize[6],
   csymacr[1],
   crname[ADDRESS];
```

field	possible values	meaning
chlang		this field specifies what the current high level language for output is; i.e. if the current level of language in use (as specified by oclang) is highlanguage, then output should be formatted to conform to the syntactical and semantical rules of this high level language
chlang	l10	the current high level language in use is L10
	cobol	the current high level language in use is COBOL
	fortran	the current high level language in use is FORTRAN
	bcpl	the current high level language in use is BCPL
	pl1	the current high level language in use is PL1
	jovial	the current high level language in use is JOVIAL
oclang		this field specifies what level of language should be used for the formatting of output
oclang	machine	the current level of language in use is machine language

field	possible values	meaning
	assembly	the current level of language in use is assembly language
	highlanguage	the current level of language in use is the current high level language
cradix	a number	all numeric output should be formatted as numbers in the base specified by this field
ctmode	this field specifies what the current output mode is	
ctmode	tmcurlang	user output should be formatted according to the current language specifications of fields oclang and ohlang
	trascii	user output should be formatted as ascii values
	tmsixbit	user output should be formatted as sixbit values
	tmrad50	user output should be formatted as radix 50 values
	tmfloat	user output should be formatted as floating point numbers
	trbyte	user output should be formatted as successive bytes, with each byte having a bytesize as specified by obytesize
	tmnumeric	user output should be formatted numerically as numbers in the base specified by oradix
	tmstring	user output should be formatted strings conforming to string data types of the current high level language

field	possible values	meaning
	tmrecord	user output should be formatted as instances of the record named by the string pointed to by ornames, conforming to record data types of the current high level language
	tmlist	user output should be formatted as lists conforming to list data types of the current high level language
	tmarray	user output should be formatted as arrays conforming to array data types of the current high level language
	tmequal	this output mode means to tell the user the numeric value of input address lists
	tmquestion	this output mode means to tell the user in which block or input symbol (as part of an address list) is defined
bytesize	a number	bytesize to use if the current output mode is tmbyte
symadr	BOOLEAN	if this field is TRUE, then display addresses as a symbol plus an offset (as discussed elsewhere); if this field is FALSE, then display addresses numerically
ornname	address	the L10 string pointed to by this field is the name of the record descriptor to be used if the current output mode is tmrecord

COROUTINES 13

Many of the routines used in various modules are coroutines (as opposed to procedures). All coroutines conform to debugger standards, i.e. LIB standards, with respect to invocation, argument passing and result returning, and general flow control. In addition to these debugger formats, most external coroutines conform to the following usage conventions: 13a

When they are CREMPORTed they are passed some (or no) arguments that remain valid for this instance of the coroutine;

The PORT ENTRY code for a coroutine does some initialization code (e.g. opening of other ports) that is valid for this instance of the coroutine and then does its EXIT PCALL;

No arguments are returned to the owning routine in the EXIT PCALL;

The results specified in the EXIT PCALL (or a terminating PCALL) phrase become the arguments for the first (or nTH) cycle of the coroutine;

(A cycle is considered to start after the EXIT PCALL or after the PCALL that terminates a cycle; a cycle is considered to terminate when the coroutine (or some routine on the coroutine's behalf) does a PCALL to the coroutine's owner with the first argument returned being 0.)

Usually, one of the arguments for a coroutine is the address of an output string to be filled in with one line of information for the user;

The coroutine writes this output string with one line of information and then PCALLs its owner.

The value of the first argument in this returning PCALL is interpreted in the following manner:

If the value is greater than 0, then the output string should have a carriage-return linefeed sequence appended to it and then it should be presented to the user; the coroutine has not completed a cycle yet and expects to be PCALLed again (with no new arguments) to continue its operation. This type of return will be called a positive return.

If the value is less than 0, then the output string should

have a space, followed by the assignment operator character, followed by several spaces, appended to it and then it should be presented to the user: the coroutine has not completed a cycle yet and expects to be PCALLed again to continue its operation; however, in this case, the coroutine usually expects to get 2 new arguments returned as results of this PCALL. This type of return will be called a negative return.

These 2 new arguments usually consist of the address of a (potentially NULL) new value string (or FALSE if the user did not specify a new value), and the address of a current input mode record to be used to interpret the new value string.

If the value is equal to 0, then the coroutine has completed a cycle; if the output string has a non-zero length, then it (the output string) is considered to contain an error message to be presented to the user. At this time, the coroutine is ready to accept new arguments to start a new cycle. This type of return will be called a terminating return or a 0 return.

GENERATING MODULES AND THE DEBUGGER LOADER 14

The following discussion is specific for generating modules designed to run under TENEX. However, the principles involved are the same regardless of what operating system the DD will be run under. 14a

Backend modules are TENEX SSAVE files that are "GET"ted at the appropriate time. 14b

The basic sequence of events to generate a module is (TENEX EXEC commands are in upper case): 14c

GET the debugger loader

REENTER the debugger loader

give the debugger loader command to start loading at the address appropriate for the module being generated

load the relocatable binary (rel) files that comprise the module being generated

terminate loading

if desired, enter TENEX DDT and perform any pre-saving initialization desired, e.g. saving the symbol table pointer in the dispatch table for the module

SSAVE the proper pages for the module on the appropriately named file

The debugger loader is a version of the SAFELDR subsystem that has been used to load several rel files of debugger-wide importance and then saved away, thus making the definitions of those items defined in these pre-loaded files available to all modules. The debugger loader contains the definitions for the following (see the appendix for a detailed description of these definitions): 14d

debugger-wide definitions,

the L10 runtime environment (for the debugger dispatcher and any other modules written in L10), and

the debugger frontend to backend communication package.

In practice, it is rather simple to make a version of the debugger. A user merely edits the NLS file CONFIG.NLS, which contains certain

self-evident switches indicating which type of debugger is to be constructed. (Currently the user can indicate whether or not a standalone version of the debugger is desired, i.e., by indicating that only one GSM and only one LM is desired - and this is the case for the L10 and JOVIAL versions of DAD.) The user then runs the TENEX program, RUNFIL, and specifies as the input file either DAD.RUN (to make an L10 version of DAD) or JDAD.RUN (to make a JOVIAL version of DAD). These runfiles will cause the submission of a batch job that will take advantage of the NLS LIBRARY Subsystem to compile and load the appropriate version of the debugger. 14e

PROGRAMMERS' GUIDE TO THE DEBUGER DISPATCHER 15

The debugger dispatcher (DD) is that module of the debuger that is responsible for communication with the debugger frontend and for dispatching user requests (made via the debugger frontend) to the appropriate routines in language and/or operating system modules (LMs and OSMs). 15a

When a user requests a specific debugging function (though her interaction with the CLI (Command Language Interpreter) and the debugger grammar), the CLI translates the user's request to a request on the debugger backend. The debugger dispatcher is that module in the debugger backend that will receive this request. The DD will then call the appropriate routine(s), in the proper LM, OSM, and/or DD, to perform the specific action and possibly to obtain results that will then be returned to the CLI to be presented to the user. 15b

PROCESS STRUCTURE AND OPERATION OF THE DEBUGGER DISPATCHER 16

The debugger dispatcher consists basically of a dispatch table, routines that will be called by the CLI, routines and data structures that will be called and referenced by other modules of the debugger (hereafter referred to as external routines and data structures), and any routines and data structures (hereafter referred to as support routines and data structures) needed for the support of the above routines and data structures. 16a

(The DD also contains the code for the support of the communication protocols for frontend-backend communication, the runtime environment for the highlevel language used by the debugger itself, and certain global data declarations that are always available to all modules of the debugger.)

User requests fall roughly into the following categories: 16b

1) requests for actions that are both language and operating system independent that cause a state change in the debugger, e.g. the user has been debugging using language X, and hence the X LM has been loaded, and now the user wishes to use language Y, and thus the Y LM must be loaded (after giving the X LM a chance to clean up);

2) requests that are both language and operating system independent, e.g. what is my current debugging state: and

3) requests that are language and/or operating system dependent, e.g. display the contents of cell N of the current target process in the current high level language.

Each of these types of requests are handled by the DD in a slightly different manner. 16c

(There is no formalization within the debugger with respect to request types. This discussion of request types is merely a mechanism for documentation communication purposes to give a feeling for how the debugger works.)

(In fact what happens is the following: a user makes a request via the CLI, and the CLI then calls the relevant procedure(s) in the debugger backend. The backend procedures then dispatch the request. Various user requests may call the same backend procedure, or a single user request may call several backend procedures.)

When the DD receives a type 2 request from the CLI, it performs whatever action is necessary to satisfy the request and then returns to the CLI, potentially returning strings to be displayed to the user. By the very nature of type 2 requests, the DD is able to satisfy these requests without calling any routines in a LM or OSM. After the DD has returned to the CLI, the type 2 request can be considered to have been completely satisfied.

When the DD receives a type 3 request from the CLI, the following typical sequence of events will occur:

the DD will look in the current appropriate (LM or OSM) dispatch table to determine if this type of request is supported by the current LM or OSM.

If this request is not supported, an appropriate error message string will be generated and returned to the CLI to be presented to the user, and this request can be considered over.

the DD will perform some syntactical and semantical checks on the arguments for this request and will convert them to internal debugger format.

If the arguments are invalid or illegal, then the DD will either generate an appropriate error message string and return to the CLI (and this request can be considered over), or the DD will interact with the user (via the CLI) to get valid arguments (and the request will proceed normally).

the DD will then invoke the appropriate LM or OSM routine(s), whose address was obtained from the appropriate dispatch table, to satisfy the request. (see the discussion elsewhere for invoking sequences.)

the invoked routine will perform its function and then return to the DD, potentially returning strings of information (which may be error messages) to be passed on to the user.

the DD will then return to the CLI, passing along any strings generated by the invoked routine, and this request can be considered over.

(In fact, if the invoked routine is a coroutine, the above 2 steps may be repeated a number of times before the request is finished.)

When the DD receives a type 1 request from the CLI, a combination of the above is likely to happen. A typical type 1 request might be that the user wishes to change the high level language that is being used as the current implementation language. This request would cause the current LM to be unloaded, and a new LM to be loaded. But before a LM can be unloaded, its termination routine (if one exists) must be called; and as soon as a new LM is loaded, its initialization (if one exists) routine must be called. However, the actual loading and unloading of the LMs are functions handled entirely by the DD.

An important part of the debugger dispatcher is its dispatch table. The dispatch table contains: 16c

addresses of external routines, and

addresses of external data structures, and

in some instances, a dispatch table entry is itself an external data structure. (A dispatch table entry that is itself a data structure will be called a simple data structure.)

To perform its function, a LM or OSM routine may find it necessary to call routines provided by the DD, or to reference data structures in the DD. To do so, the LM or OSM routine will use the DD dispatch table and can thus call or reference routines and/or data structures that it does not provide itself. 16e

GENERATING THE DEBUGGER DISPATCHER 17

The following discussion is specific for generating the debugger dispatcher module designed to run under TENEX. However, the principles involved are the same regardless of what operating system the DD will be run under. 17a

The sequence of events to generate a DD is (TENEX EXEC commands are in upper case): 17b

using NLS, edit the file CONFIG.NLS to insure that the switches have the correct value

at the TENEX EXEC, run the program RUNFIL, specifying as the input file the file LOADFE.RUN -- this will properly compile and load the debugger FE

at the TENEX EXEC, run the program RUNFIL, specifying as the input file the file LOADLDR.RUN -- this will properly compile and generate the debugger loader

at the TENEX EXEC, run the program RUNFIL, specifying as the input file the file LOADDD.RUN -- this will properly compile and generate the debugger DD module

THE DD DISPATCH TABLE

Many of the entries in the DD's dispatch table are copies of entries in the OSM's dispatch table, or pointers into the OSM's dispatch table. These entries are copied from the OSM's dispatch table after the OSM is initialized. This is done so that LM routines do not have to know about the OSM dispatch table and thus only have to deal with one dispatch table, i.e., the DD's dispatch table.

The symbolic offset names for the entries in the DD dispatch table are contained in the file DDTOSP.NLS. Also, the debugger loader contains these definitions. (Note that an offset of 0 refers to the first entry in the dispatch table.)

decimal offset	symbolic offset name	meaning
0	ddiosi	see OSM osini
1	ddsymp	see OSM ossymc
2	ddtdwosm	see OSM ostdwosm
3	ddrnt	see OSM osrnt
4	ddtirit	see OSM ostinit
5	ddtowtl	see OSM ostdwtl
6	ddtrrtr	see OSM ostrntr
7	ddbpte	see OSM osbpte
8	ddmsta	see OSM osmsta
9	ddrd1w	see OSM osrd1w
10	ddrdrw	see OSM osrdnw
11	ddstat	see OSM osstat
12	ddwr1w	see OSM oswr1w
13	ddwrrw	see OSM oswrnw

decimal offset	symbolic offset name	meaning
14	ccsrcm	see OSM ossrcm
15	ccgpfs	see OSM osgpfs
16	ccspfs	see OSM osspfs
17	ccalcs	see OSM osalos
18	ccrels	see OSM osrels
19	ccrmbrk	see OSM osrmbrk
20	ccinbrk	see OSM osinbrk
21	cccsacr	see OSM oscsacr
22	ccgpc	see OSM osgpc
23	ccostc	see OSM osdntl
24	ccstop	see OSM osstop
25	ccgo	see OSM osgo
26	ccpdcsm	see OSM ospdwcsm
27	ccesta	see OSM osesta
28	ccfsta	see OSM osfsta
29	ccfsav	see OSM osfsav
30	ccosidn	see OSM osidn
31	ccsvec	see OSM ossvec
150	cclarc	address of the current LM's dispatch table
151	ccmarc	maximum offset value for symbolic address displays

decimal offset	symbolic offset name	meaning
152	ddlste	value of the last evaluated address range element
153	ddddca	address of DD's decode address range routine
154	ddchrt	address of GFS data structure
155	ddchrs	address DD routine for changing GFS data structure
156	ddarng	address DD routine to manage LST+DIS data structure
157	ddlsval	last displayed value
158	ddsfmk	default search mask
159-161	----	RESERVED FOR FUTURE USE
162	ddclm	current LM
163	ddddsm	current DSM
164	ddatltbl	address of tool table
165	ddhrtrd	address of DIREC chain
166	ddctool	address of current tool TOOLREC
167	ddcprc	address of current fork TOOLREC
168	ddich	highest used IDH
169	ddgtirc	address of procedure to get a tool record
170	ddgich	address of procedure to get an IDH
171	ddxcch	address of XCREC chain
172	ddfrxc	address of procedure to free an XCREC record
173	ddkprcc	address of procedure to kill a process

decimal offset	symbolic offset name	meaning
174	ddpchrstr	address of procedure to printing representation for a character
175	ddgetdesc	address of procedure to get DEC character for a function
176	ddgetbrka	address of procedure to get the address at which a breakpoint is set

DETAILED DISCUSSION OF EACH ENTRY IN THE DD'S DISPATCH TABLE 19

This section will discuss in detail each entry in the DD's dispatch table. Each entry will be discussed under its symbolic offset name. Those entries that are copies of entries in the OSM dispatch table will not be discussed here, but are discussed in the appropriate section of this manual later on. 19a

ddlanc 19b

entry type - address of a data structure

data structure name - LANDSP

data structure meaning -

this is the address of the language module's dispatch table

data structure type -

this data structure is composed of 50 words

ddacro 19c

entry type - single data structure

data structure name - MAXOFFSET

data structure meaning -

if addresses are being displayed as a symbol plus an offset, then if the offset is greater than the value of this cell, the address should be displayed numerically.

data structure type -

this data structure consists of the single word in the DD dispatch table

dolste 19d

entry type - simple data structure

data structure name - LSTEADR

data structure meaning -

the value of this data structure is the value of the last completely evaluated address range element.

data structure type -

this data structure consists of the single word in the DD dispatch table

discussion -

This data structure should be updated by the LM every time it evaluates an address range element for which it is meaningful to update this cell (e.g. it is not meaningful to update this cell after the evaluation of an ARE that corresponds to the target process' signal status).

ddddda 19e

entry type - procedure address

procedure function (brief) -

This procedure is used to determine the gross type of AREs.

when called -

This procedure will be called by any DD, LM, or OSM routine that evaluates AREs to determine the gross type of the ARE before it is evaluated.

arguments -

1st argument: the address of the first ARE string

2nd argument: the address of the corresponding second ARE string

results -

1st result:

a value indicating the gross type of the address range

error conditions -

this procedure will return an illegal gross type on any errors that it detects

discussion -

Before any DD, LM, or OSM routine completely evaluates an ARE, this routine must be called to determine the gross type of the ARE. This procedure thus provides for uniform interpretation of AREs. If an ARE is illegal or invalid, then this procedure will return a gross type indicating this.

ddchrt 19f

entry type - address of a data structure

data structure name - GFS

data structure meaning -

this is the DD Generic Function String

data structure type -

this is a 128 character LIO string

ddchrs 19g

entry type - procedure address

procedure function (brief) -

This procedure is used to modify the GFS.

when called -

If a LM wishes to change which character will be used for which generic function (e.g. at initialization time), the LM must use this procedure and NOT modify the GFS directly.

arguments -

1st argument: the address of a string containing as its first character the ascii character that is to perform a generic function

2nd argument: the generic function the character is to serve

3rd argument: zero or the address of a result list which will be filled in with the first element of the list being set to the OFS

results -

If a third argument is specified, then on success the first element of the result list (whose address is passed as the non-zero third argument) will get a copy of the OFS; if the third argument is zero, then nothing will be returned on success. If a third argument is passed, and this routine detects bad first or second arguments, then this routine will generate a L1D HELP signal in an attempt to get correct arguments; if no help is available, or if no third argument is passed, then a L1D ABORT signal will be generated.

error conditions -

this procedure will generate an L1D ABORT signal if it receives bad input.

discussion -

This procedure is used to modify the OFS data structure.

dcarno 19h

entry type - procedure access

procedure function (brief) -

This procedure is used for reading or writing the DD data structure LSTADIS (which contains the access of the last n displayed cells).

when called -

This procedure will be called whenever any DD, LV, or OSM routine wishes to read or write the LSTADIS data structure.

arguments -

1st argument:

FALSE to indicate read an entry from the LSTADIS data structure; TRUE to make a new entry in LSTADIS.

2nd argument:

if this is a read operation, then this argument is the index of the last displayed address desired, e.g. the most recently displayed access has an index of 0, the address displayed before that has an index of 1, etc.; if this is a write operation, then this is the new address to add to LSTADIS

results -

for write operations -

NONE

for read operations -

the index-th (mod n, where n is the number of entries maintained, and is currently set to 4) last displayed address

error conditions -

NONE

discussion -

this procedure is used to maintain the LSTADIS data structure.

ddlcval 191

entry type - single data structure

data structure name - LSTVDIS

data structure meaning -

this is the value of the last displayed cell

data structure type -

this data structure consists of the single word in the DD dispatch table

discussion -

this data structure should be maintained by LM routines wherever they display cells to the user.

ddcfmk 19j

entry type - single data structure

data structure name - DEFMASK

data structure meaning -

this is the debugger default mask for content searches and memory setting commands

data structure type -

this data structure consists of the single word in the DD dispatch table

discussion -

The LM lnmass routine is responsible for setting this cell; and the LM routines lnfmem and lnmems may use this cell

ddclm 19k

entry type - single data structure

data structure meaning -

this data structure is an indicator of which LM is current

data structure type -

this data structure consists of the single word in the DD dispatch table

discussion -

This cell also indicates which LM is currently loaded and operational

ddddsm 19l

entry type - simple data structure

data structure meaning -

this data structure is an indicator of which OSM is current

data structure type -

this data structure consists of the single word in the DD dispatch table

discussion -

This cell also indicates which OSM is currently loaded and operational

ddatltbl 19m

entry type - simple data structure

data structure meaning -

this cell points to the chain of tool records

data structure type -

this data structure consists of the single word in the DD dispatch table

discussion -

for each process that the debugger knows about, a record known as the tool record is maintained. This records are chained together to model any process structure inherent in the processes themselves. This entry in the DD dispatch table points to this chain of tool records. (see DDTDEF for the definition of the tool records.)

ddhrtrc 19n

entry type - simple data structure

data structure meaning -

this cell points to the chain of DIREC records

data structure type -

this data structure consists of the single word in the DD dispatch table

discussion -

Whenever the debugger is entered, a Debugger Instance RECord (DIREC) is created and chained to previously existing DIRECs. This is one of the mechanisms the debugger uses to maintain its own status. This is necessary because the debugger may be entered recursively (e.g., the user executes some instructions while in the debugger because of a breakpoint; during the execution of those instructions, another breakpoint may be encountered.) These records are deleted when the user resumes execution of the process that caused the corresponding DIREC to be created. (see DDTDEF for a definition of the DIREC record).

ddctool 19o

entry type - simple data structure

data structure meaning -

this cell points to the tool record for the top process in a group of related processes, i.e., a tool

data structure type -

this data structure consists of the single word in the DD dispatch table

discussion -

a group of related processes forming a process tree is referred to as a tool; this cell points to the tool record of the process that represents the top process in a tool.

ddcpro 19b

entry type - simple data structure

data structure meaning -

this cell points to the tool record for the current process

data structure type -

this data structure consists of the single word in the DD dispatch table

discussion -

this cell points to the tool record for the current process.

ddidh 19c

entry type - simple data structure

data structure meaning -

this cell contains the highest used IDH

data structure type -

this data structure consists of the single word in the DD dispatch table

discussion -

each time the debugger learns of a new process it assigns that

new process or IDM; this data structure is used to remember what the highest used IDM is.

dcgtlrc 19r

entry type - procedure address

procedure function (brief) -

This procedure is used to obtain a clean tool record.

when called -

Whenever the debugger needs a fresh tool record, it calls this procedure.

arguments -

NONE

results -

1st result:

the address of a free tool record

error conditions -

NONE

discussion -

dcgidh 19s

entry type - procedure address

procedure function (brief) -

This procedure is used to obtain a new IDH.

when called -

Whenever the debugger needs a fresh IDH, it calls this procedure.

arguments -

NONE

results -

1st result:

a new IDH

error conditions -

NONE

discussion -

This routine maintains the dcidh data structure

ddxcch 19t

entry type - simple data structure

data structure meaning -

this cell points to the chain of XCREC records

data structure type -

this data structure consists of the single word in the DD dispatch table

discussion -

When the debugger executes instructions out-of-line, i.e., in response to the user issuing the EXECUTE command, a data structure called a DIREC record is maintained in order to remember certain things about the state of the world. This data structure points to a chain of such records. (see DDTDEF for a definition of the XCREC record.)

ddfrxc 19u

entry type - procedure address

procedure function (brief) -

This procedure is used to free an XCREC.

when called -

Whenever the debugger needs to free an XCREC.

arguments -

1st argument

the address of the XCREC to be freed

results -

NONE

error conditions -

NONE

discussion -

This routine returns to free storage the passed XCREC.

ackproc 19v

entry type - procedure address

procedure function (brief) -

This procedure is used whenever a process is killed (either by the user, or by other processes).

when called -

whenever the debugger needs to kill a process.

arguments -

1st argument

the address of the tool record corresponding to the process to be killed.

results -

NONE

error conditions -

NONE

discussion -

This routine will cleanup as necessary to remove all knowledge

that the debugger has about a process that is about to disappear. It will use the appropriate LM and OSM routines as well.

dcpchrstr 19w

entry type - procedure address

procedure function (brief) -

This procedure is used to get the printing representation of certain control characters.

when called -

As needed for formatting.

arguments -

1st argument

the character whose printing representation is desired.

2nd argument

the address of a string to get the printing representation of the passed character.

results -

NONE

error conditions -

NONE

discussion -

This routine will return the printing representation of the passed character in the passed string.

dcgetgesc 19x

entry type - procedure address

procedure function (brief) -

This procedure is used to get the character which is currently serving a specific generic function.

when called -

Whenever any debugger routine needs to know the current character that is serving a specific generic function.

arguments -

1st argument

the generic function whose character is desired

results -

NONE

error conditions -

NONE

discussion -

This routine will return the character which is currently serving the passed generic function.

dggetbrka 19y

entry type - procedure address

procedure function (brief) -

This procedure is used to get the address at which a specific breakpoint is set.

when called -

Wherever any debugger routine needs to know the address at which a breakpoint is set.

arguments -

1st argument

the breakpoint number for which it is desired to know the address at which the breakpoint is set.

results -

0 or the address at which the passed breakpoint is set

error conditions -

NONE

discussion -

19z

GENERAL DISCUSSION OF ROUTINES CALLABLE FROM THE CLI 20

The CLI will in response to user interactions, call procedures in the debugger backend. This document will not go into any detail about the communication protocol used for this purpose. Suffice it to say, that the communication part of the DD is well isolated and easily changeable. Calls from the CLI get translated into debugger formatted calls, with debugger formatted arguments, by this communication code within the DD. Calls from the debugger backend to the CLI, and results from calls on the debugger backend have a similar inverse translation applied to them. 20a

The following section will discuss the routines in the DD that are callable from the CLI in terms of the debugger call/return mechanisms. 20b

(Note that all routines that are called by the CLI accept as their *last argument the address* of a result list. It is this list that gets the results discussed below; i.e. the first result is actually returned as the first element of the passed result list, the second result is actually returned as the second element of the passed result list, etc.)

DETAILED DISCUSSION OF ROUTINES CALLABLE FROM THE CLI 21

procedure name - initdt 21a

procedure function (brief) -

the function of this routine is to initialize the debugger

arguments -

1st argument:

the value of this argument indicates which operating system module should be loaded

2nd argument:

the value of this argument indicates which language module should be loaded

3rd argument:

the address of the result list

results -

1st result:

the DFS is returned to the CLI

error conditions -

If this procedure cannot perform its functions it will generate an ABORT to the CLI.

discussion -

This procedure performs all first time initialization required to run the debugger. This includes the initialization of the input and output mode records, the initialization of the DFS, the loading and initialization of the initial LM and DSM, and the establishment of a communication path from the debugger backend to the CLI.

Upon successful completion of its tasks, it will return a copy of the DFS to the CLI.

procedure name - xcbgp 21b

procedure function (brief) -

the function of this procedure is to establish the state necessary for the debugging of a process, i.e. to make the passed process the target process.

arguments -

1st argument:

process name

2nd argument:

the address of the result list

results -

NONE

error conditions -

This procedure will generate an ABORT (with an appropriate message) to the CLI if it is not possible to establish the debugging state necessary to debug the passed process name.

discussion -

this procedure will first convert the passed process name into the internal debugger handle for the process.

Next it will call the terminate debug process routines in the CSV and then in the LM.

And finally, it will call the breakpoint enter routines in the CSV and then the LM, indicating that a new process has been specified.

The called routines are expected to perform whatever cleanup and/or initialization is required to perform debugging.

procedure name - xcchrc 21c

procedure function (brief) -

arguments -

1st argument:

the address of the result list

results -

NONE

error conditions -

NONE

discussion -

this procedure will interpret the GFS and generate a number of lines of information. Each line consists of user readable information pertinent to successive characters in the GFS. After each line has been formatted a call on the utility routine prtstr will be made to present the line to the user. A normal return will be given to the CLI when all such lines have been formatted and presented to the user.

procedure name - xcchrs 21d

procedure function (brief) -

to modify the GFS

arguments -

see entry points in the detailed discussion of the DD's dispatch table

results -

see entry points in the detailed discussion of the DD's dispatch table

error conditions -

see entry points in the detailed discussion of the DD's dispatch table

discussion -

This is the procedure whose address is contained in entry points of the DD's dispatch table. This procedure can be called by the CLI in response to user request to modify the GFS, as well as being able to be called by LM initialization routines.

procedure name - xdtycd 21e

procedure function (brief) -

tc interpret and display to the user the permanent output mode record

arguments -

1st argument:

the address of the result list

results -

NONE

error conditions -

NONE

discussion -

this procedure will interpret the permanent output record and generate lines of information (reflecting this interpretation), and make use of the utility routine pntstr to present these lines to the user.

procedure name - xctypo 21f

procedure function (brief) -

to modify the permanent output mode record

arguments -

1st argument:

a value representing the main output mode keyword specified by the user

2nd argument:

a value representing the secondary output mode keyword specified by the user if the main output mode has a secondary mode; FALSE otherwise

3rd argument:

the address of the result list

results -

NONE

error conditions -

if this routine is passed illegal or invalid arguments it will do nothing

discussion -

this procedure is called by the CLI in response to a user's request to change the permanent output mode

procedure name - xcinpc 21g

procedure function (brief) -

tc interpret and display to the user the permanent input mode record

arguments -

1st argument:

the address of the result list

results -

NONE

error conditions -

NONE

discussion -

this procedure will interpret the permanent input mode record and generate lines of information (reflecting this interpretation), and make use of the utility routine pntstr to present these lines to the user.

procedure name - xcinps 21h

procedure function (brief) -

tc modify the permanent input mode record

arguments -

1st argument:

a value representing the main input mode keyword specified by the user

2nd argument:

a value representing the secondary input mode keyword specified by the user if the main input mode has a secondary mode; FALSE otherwise

3rd argument:

the address of the result list

results -

NONE

error conditions -

if this routine is passed illegal or invalid arguments it will do nothing

discussion -

this procedure is called by the CLI in response to a user's request to change the permanent input mode

procedure name - xcouts 214

procedure function (brief) -

to allow displayed output to be directed to the user (via his terminal) and/or to a file.

arguments -

the first 3 arguments comprise a state table as follows (see discussion below for the meanings of each of the states):

arg 1	arg 2	arg 3	state
FALSE	FALSE	--	1
FALSE	TRUE	FALSE	2
FALSE	TRUE	"FILE" (*)	3
FALSE	TRUE	"TERMINAL" (**)	4
file (***)	FALSE	--	5
file (***)	TRUE	--	6

footnotes:

(*) - this argument is the value for the keyword "FILE"

(**) - this argument is the value for the keyword "TERMINAL"

(***) - this argument is the address of a string containing the name of the file the user wishes to have his output printed on

4th argument:

the address of the result list

results -

1st result:

for requested states 5 or 6, the full file name will be returned to the CLI; for requested states 1-4, there are no results

error conditions -

if states 5 or 6 are requested and the first argument is invalid or illegal, a HELP will be generated to the CLI to attempt to make it all ok; if no help is available an ABORT will be generated.

this routine assumes that it will be called by the CLI and that only valid combinations of the arguments will be passed.

discussion -

this routine provides the user with control over where his output will be presented. Output can be displayed at her terminal and/or written in a file. (Note that if output is only being written in a file, then certain commands are no longer available to the user.)

The first 3 arguments specify which state the user wishes to be in as follows:

state	meaning
1	get back to the default state, i.e. display output on the user's terminal and close any open output files
2	an output file has been previously specified, and the user wishes output to be both displayed at his terminal and written in the output file
3	an output file has been previously specified, and the user wishes output to be written in the output file only
4	an output file has been previously specified, and the user wishes output to be displayed only on his terminal

5 the first argument specifies the name of an output file that the user wishes to have his output written on in addition to being displayed on her terminal; if the named file already exists, then the new output will be written on the end of the file

6 the first argument specifies the name of a new output file that the user wishes to have his output written in addition to being displayed on her terminal

procedure name - xcmesg 21

procedure function (brief) -

to display the default debugger mask to the user in the permanent output mode radix

arguments -

1st argument:

the address of the result string

results -

NONE

error conditions -

NONE

discussion -

this routine will use the utility routine putstr to display the default mask (DEFMASK) to the user as a number in the permanent output mode radix

procedure name - xcmass 21k

procedure function (brief) -

to set up the default mask (DEFMASK)

arguments -

1st argument:

FALSE meaning to use the permanent input mode record to interpret the third argument; or the value of the keyword the user specified for the main input mode to be used to interpret the third argument

2nd argument:

only has meaning if the first argument is not FALSE; in this case it is the value of the secondary input mode to be used for the interpretation of the third argument (or it can be FALSE if the main input mode does not require a secondary mode)

3rd argument:

the address of a string to be evaluated, according to the current input mode, to become the new default debugger mask (DEFMASK)

4th argument:

the address of the result list

results -

NONE

error conditions -

This routine will generate an ABORT (with an appropriate message) to the CLI if the current LM does not provide the lnmass routine.

Other error conditions will be displayed to the user as strings via the pntstr utility routine.

discussion -

This routine makes use of the LM lnmass routine to evaluate the third argument and to set up DEFMASK. If lnmass returns a non-null string it will be presented to the user via the utility routine pntstr.

procedure name - sadrl 211

procedure function (brief) -

to display, and optionally to assign to, address lists

arguments -

1st argument:

the address of a string containing the address list to be displayed and optionally assigned to

2nd argument:

FALSE; or value of CML keyword that user used to terminate the specified address list; or the value of the CML keyword to be used as the main output mode

3rd argument:

FALSE; or the value of the secondary output mode keyword

4th argument:

FALSE; or TRUE indicating that this should be an assignment as well as a display operation

(this will be true if the user uses the 2nd and 3rd arguments for specifying output mode, and he wants to do an assignment; otherwise this will most likely be false and assignment will be indicated, if desired, by the access list terminating character.)

5th argument:

the address of the result list

results -

NONE

error conditions -

any error conditions detected by this routine will be handled internally and an appropriate error message will be displayed to the user via the utility routine pntstr

discussion -

this is the main routine used for displaying and assigning to access lists. it breaks down the passed address list into address ranges and then uses the LM lnsacr routine to obtain strings to be presented to the user; it makes use of the utility routine pntstr to display strings to the user for the display only case, and uses the utility routine pasnstr to display strings to the user and get new values for the display and assignment case.

procedure name - xctadrl 21m

procedure function (brief) -

to display an address list only on the user's terminal (and optically to assign to the address list) independent of whether or not the user had specified that his output should go to a file and/or his terminal

arguments -

see sadrl procedure

results -

see sadrl procedure

error conditions -

see sadrl procedure

discussion -

This procedure uses sadrl to do most of its work. What it does is temporarily (for the course of displaying and assigning to the address list) modify the output state of the debugger so that output goes only to the users terminal.

procedure name - xcpadrl 21n

procedure function (brief) -

to print an address list in the user's output file

arguments -

1st argument:

see sadrl procedure

2nd argument:

see sadrl procedure

3rd argument:

see sadrl procedure

4th argument:

the address of a string containing the name of an output file; or FALSE meaning to use the existing output file

5th argument:

see sadrl procedure

results -

see sadrl procedure

error conditions -

see sadrl procedure

discussion -

This procedure uses sadrl to do most of its work. What it does is temporarily (for the course of displaying to the address list) modify the output state of the debugger so that output is written only on the file specified as the 4th argument.

procedure name - xctab 21o

procedure function (brief) -

to use the value of the data structure LSTVDIS (the last value displayed to the user) as an address list to be displayed, using the same output mode used in the previous command

arguments -

1st argument:

the address of the result list

results -

see sadrl procedure

error conditions -

see sadrl procedure

discussion -

This procedure uses sadrl to do most of its work. What it does is generate an address list string from the value in the LSTVDIS data structure and then call sadrl to do its work. It is the responsibility of the LM to maintain LSTVDIS.

procedure name - xdbound 21b

procedure function (brief) -

to use the value of the first element of the LSTADIS data structure (the last N accesses displayed to the user) as an access list to be displayed, using the same output mode used in the previous command

arguments -

1st argument:

the address of the result list

results -

see sadrl procedure

error conditions -

see sadrl procedure

discussion -

This procedure uses sadrl to do most of its work. What it does is generate an address list string from the value of the first element of the LSTADIS data structure and then call sadrl to do its work. It is the responsibility of the LV to maintain LSTADIS.

(It gets the value of the first element of the LSTADIS data structure by using the external routine adarnc.)

Basically, this routine provides an inverse for the xdtab routine; e.g. if the user had displayed n, and the contents of cell n were m, and then the user gave the command that caused the xdtab routine to be called, then cell m would be displayed; if the user then gave the command to cause the xdbound routine to be called, then cell n would be displayed again.

procedure name - xclnfc 21c

procedure function (brief) -

to display the next cell in the memory of the target process, using the same output mode used in the previous command

arguments -

1st argument:

the address of the result list

results -

see sadrl procedure

error conditions -

see sadrl procedure

discussion -

This procedure uses sadrl to do most of its work. What it does is call the LM lmnacr routine to generate an address list and then this address list is passed to sadrl.

procedure name - xcubar 21r

procedure function (brief) -

to display the previous cell in the memory of the target process, using the same output mode used in the previous command

arguments -

1st argument:

the address of the result list

results -

see sadrl procedure

error conditions -

see sadrl procedure

discussion -

This procedure uses sadrl to do most of its work. What it does is call the LM lmadr routine to generate an address list and then this address list is passed to sadrl.

procedure name - xcasgn 21s

procedure function (brief) -

to assign to the address list used in the previous command

arguments -

1st argument:

the address of the result list

results -

see sadrl procedure

error conditions -

see sadrl procedure

discussion -

This procedure uses sadrl to do most of its work. What it does is call sadrl, passing the address list used in the previous command and simulating the user terminating the address list with the LABRCWCHAR.

procedure name - xeol 21t

procedure function (brief) -

to obtain the value of the address list used in the previous command

arguments -

1st argument:

the address of the result list

results -

see sadrl procedure

error conditions -

see sadrl procedure

discussion -

This procedure uses sadrl to do most of its work. What it does is call sadrl, passing the address list used in the previous command and simulating the user terminating the address list with the EQUALCHAR.

procedure name - xcexcm 21u

procedure function (brief) -

to display the address list used in the previous command in ascii output mode

arguments -

1st argument:

the address of the result list

results -

see sadrl procedure

error conditions -

see sadrl procedure

discussion -

This procedure uses sadrl to do most of its work. What it does is call sadrl, passing the address list used in the previous command and simulating the user terminating the address list with the EXC*ARKCHAR.

procedure name - xcslsh 21v

procedure function (brief) -

to display the access list used in the previous command in symbolic output mode

arguments -

1st argument:

the address of the result list

results -

see sadrl procedure

error conditions -

see sadrl procedure

discussion -

This procedure uses sadrl to do most of its work. What it does is call sadrl, passing the address list used in the previous command and simulating the user terminating the address list with the SLASHCHAR.

procedure name - xcbslsh 21w

procedure function (brief) -

to display the address list used in the previous command in string output mode

arguments -

1st argument:

the address of the result list

results -

see sadrl procedure

error conditions -

see sadrl procedure

discussion -

This procedure uses sadrl to do most of its work. What it does is call sadrl, passing the address list used in the previous command and simulating the user terminating the address list with the BSLASHCHAR.

procedure name - xclsc 21x

procedure function (brief) -

to display the address list used in the previous command in numeric output mode

arguments -

1st argument:

the address of the result list

results -

see sadrl procedure

error conditions -

see sadrl procedure

discussion -

This procedure uses sadrl to do most of its work. What it does is call sadrl, passing the address list used in the previous command and simulating the user terminating the address list with the LSCLARECHAR.

procedure name - xdrsa 21y

procedure function (brief) -

to display the address list used in the previous command in record output mode

arguments -

1st argument:

the address of the result list

results -

see sadrl procedure

error conditions -

see sadrl procedure

discussion -

This procedure uses sadrl to do most of its work. What it does is call sadrl, passing the address list used in the previous command and simulating the user terminating the address list with the RSGUARECHAR.

procedure name - xcgmark 21z

procedure function (brief) -

to display the blocknames for the symbols used in the access list used in the previous command in string output mode; i.e. this routine provides a mechanism to determine in which blocks symbols are defined

arguments -

1st argument:

the address of the result list

results -

see sadrl procedure

error conditions -

see sadrl procedure

discussion -

This procedure uses sadrl to do most of its work. What it does is call sadrl, passing the access list used in the previous command and simulating the user terminating the address list with the GMARKCHAR.

procedure name - xcfind 21aa

procedure function (brief) -

tc display, and optionally assign to, those cells within the bounds of an address list that meet specific requirements

arguments -

1st argument:

the value of the CML keyword indicating whether this is to be a search for references to the passed 2nd argument, or a search for cells whose value is specified by the 2rd argument, or a search for cells whose value is not equal to the 2nd argument

2rd argument:

the address of a string to be evaluated, according to the current input mode, to serve as the search argument

3rd argument:

FALSE or a value representing the main current input mode keyword specified by the user

4th argument:

FALSE or a value representing the secondary current input mode keyword specified by the user if the main input mode has a secondary mode

5th argument:

mask specification as follows: if FALSE then use default mask (DEFMASK); otherwise the address of a string to be evaluated, according to the current input mode, to be used as the mask

6th argument:

the address of a string containing the address list to be searched, displayed, and optionally assigned to

7th argument:

FALSE: or value of CML keyword that user used to terminate the specified address list; or the value of the CML keyword to be used as the main output mode

8th argument:

FALSE; or the value of the secondary output mode keyword

9th argument:

FALSE: or TRUE indicating that this should be an assignment as well as a display operation

10th argument:

the address of the result list

results -

NONE

error conditions -

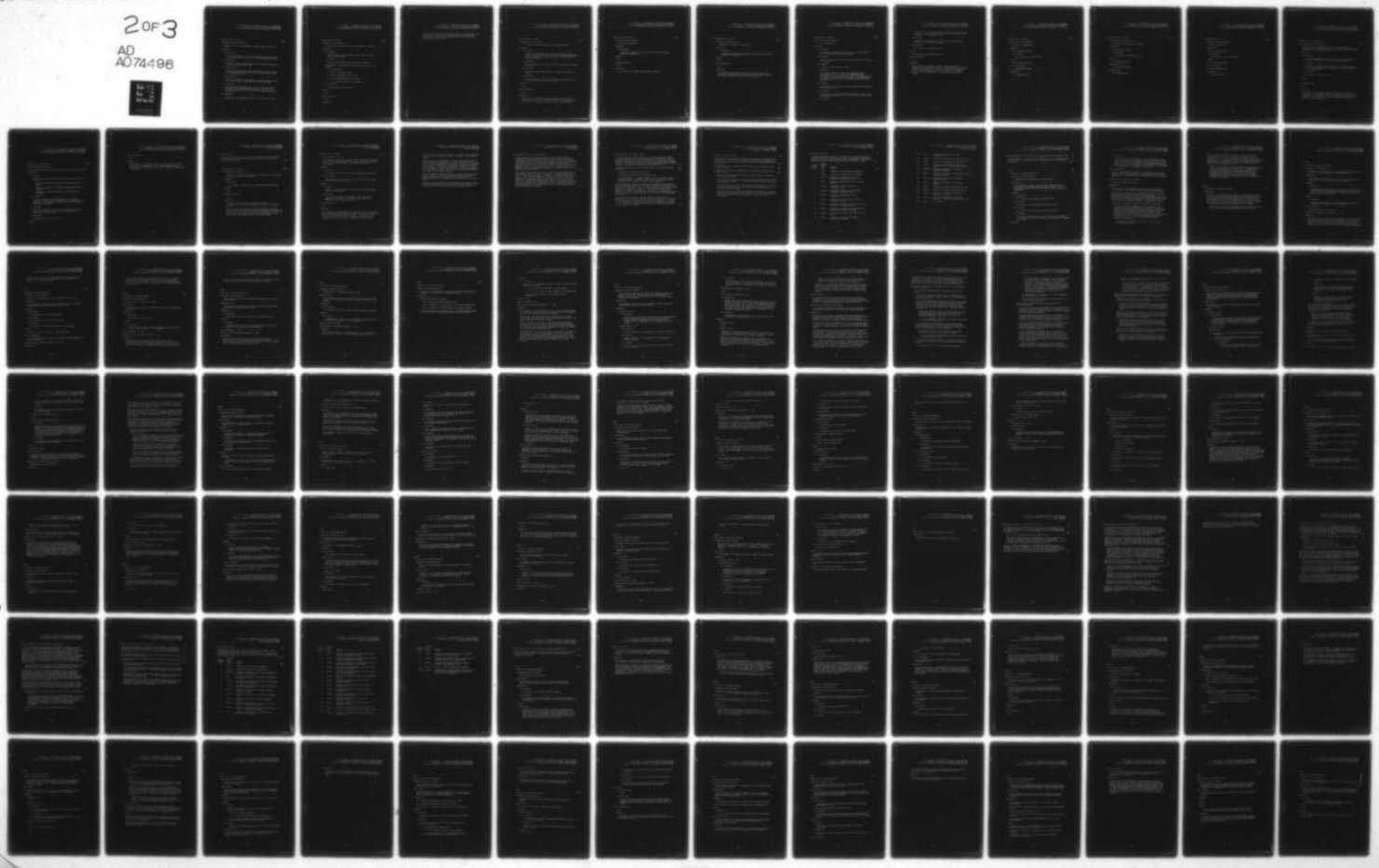
any error conditions detected by this routine will be handled internally and an appropriate error message will be displayed to the user via the utility routine putstr

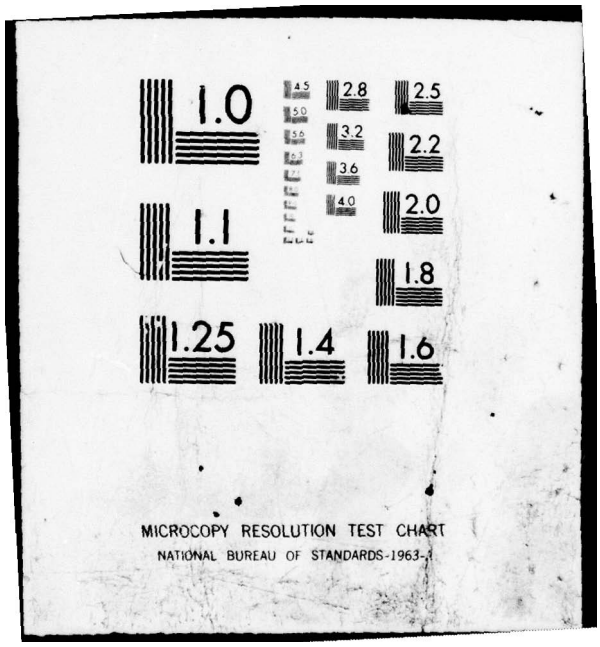
discussion -

This routine breaks down the passed address list into address ranges and then uses the LM infmem routine to obtain strings to be presented to the user; it makes use of the utility routine prtstr to display strings to the user for the display only case, and uses the utility routine passnstr to display strings to the user and get new values for the display and assignment case.

AD-A074 496 SRI INTERNATIONAL MENLO PARK CA F/G 9/2
ON-LINE PROGRAMMER'S MANAGEMENT SYSTEM. ADDENDUM II. PROGRAMMER--ETC(U)
AUG 79 B L PARSLEY, H G LEHTMAN, S KAHN F30602-77-C-0185
UNCLASSIFIED RADC-TR-79-205-ADD-2 NL

2 OF 3
AD
A074496




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

procedure name - xcmems 21ab

procedure function (brief) -

to set all cells in a specified address list to a specific value

arguments -

1st argument:

the address of a string to be evaluated, according to the current input mode, to serve as the value to set memory to

2nd argument:

FALSE or a value representing the main current input mode keyword specified by the user

3rd argument:

FALSE or a value representing the secondary current input mode keyword specified by the user if the main input mode has a secondary mode

4th argument:

FALSE to indicate no mask should be used when modify memory; TRUE means get mask specification from 5th argument

5th argument:

mask specification as follows: if FALSE then use default mask (DEFMASK); otherwise the address of a string to be evaluated, according to the current input mode, to be used as the mask

6th argument:

the address of a string containing the address list whose memory is to be modified

procedure name - xcstac 21ad

procedure function (brief) -

tc display the current state of the debugger to the user

arguments -

1st argument

FALSE to imply a short status report; TRUE to imply a verbose status report

2nd argument

process specification as per the next argument

3rd argument - an indicator of what the 2nd argument is as follows:

0 - all processes

1 - only the current process

2 - 2nd argument specifies an IDH

3 - 2nd argument specifies a use name

4 - 2nd Argument specifies a local name

4th argument:

the address of the result list

results -

NONE

error conditions -

NONE

discussion -

this routine will give the user the status of the debugger and one or more processes as specified by the arguments; this routine will use the utility routine pntstr for presenting status results to the user

procedure name - xcspcs 21ae

procedure function (brief) -

to modify the execution speed of the target process

arguments -

1st argument

FALSE - to return to normal execution speed; LANGUAGE to indicate single step at the high level language instruction level; MACHINE to indicate single step at the machine instruction level

2nd argument

TRUE to indicate to deal with called procedures as a single instruction

3rd argument

TRUE to indicate execute until a transfer instruction is executed

4th argument

TRUE to indicate to continue execution automatically after executing one of the above type breaks

results -

NONE

error conditions -

NONE

discussion -

this routine will set the execution speed of the current process to either normal execution speed or to the specified mode of single stepping, execution until a transfer, etc.

procedure name - xcbrkc 21af

procedure function (brief) -

tc remove a breakpoint

arguments -

1st argument

breakpoint number to be cleared or zero (0) to indicate clear all breakpoints

results -

NONE

error conditions -

NONE

discussion -

this routine will remove the indicated breakpoint

procedure name - xdbrkd 21ag

procedure function (brief) -

to display the status of a breakpoint

arguments -

1st argument

breakpoint number to be displayed or zero (0) to indicate display all breakpoints

results -

NONE

error conditions -

NONE

discussion -

this routine will display (to the user) the status of the indicated breakpoint; it will use the utility routine prtstr for actually presenting information to the user

procedure name - xcbrks 21ah

procedure function (brief) -

to set a breakpoint

arguments -

1st argument

the breakpoint number to be set or zero (0) to let the debugger choose an appropriate number

2nd argument

the adr of a string containing the address at which to set the breakpoint

3rd argument

currently unused - should be zero

4th argument

an indication of how to treat the breakpoint upon encountering it, i.e., enter the debugger, proceed automatically after notifying the user of encountering the breakpoint, or only enter the debugger after encountering the breakpoint for the nth time as specified by the next argument

5th argument

the proceed count if the 4th argument specified this mode; otherwise zero (0)

6th argument

the address of a string containing debugger commands to be executed if this breakpoint is taken (i.e., control enters the debugger)

7th argument

the address of a string containing debugger commands to be executed if this breakpoint is not taken (e.g., the proceed count has not yet reached zero)

8th argument

zero (0) or the address of a string containing a user supplied name for this breakpoint

9th argument:

the address of the result list

results -

the breakpoint number actually set

error conditions -

NONE

discussion -

this routine will actually establish the breakpoint as specified by the passed arguments; it will use the LM lnevbaor routine for evaluating the string containing an address at which to set the breakpoint; it will use the OSM osrmbrk routine if this breakpoint was previously set at another address.

procedure name - xctrpc 21af

procedure function (brief) -

to remove a tracepoint

arguments -

to be specified later

rth argument:

the address of the result list

results -

to be specified later

error conditions -

to be specified later

discussion -

NOT IMPLEMENTED YET

procedure name - xctrcp 21a

procedure function (brief) -

to display the status of a tracepoint

arguments -

to be specified later

1th argument:

the address of the result list

results -

to be specified later

error conditions -

to be specified later

discussion -

NOT IMPLEMENTED YET

procedure name - xctrcs 21ak

procedure function (brief) -

tc set a tracepoint

arguments -

tc be specified later

rth argument:

the address of the result list

results -

tc be specified later

error conditions -

tc be specified later

discussion -

NOT IMPLEMENTED YET

procedure name - xccontinue 21a1

procedure function (brief) -

to resume target process execution after encountering a breakpoint or a tracepoint or to start target process execution after specifying a target process

arguments -

1st argument

FALSE or the address of a string containing an address in the target process at which to continue execution

2nd argument

FALSE to indicate actually continue the process, or other to indicate to merely set the address at which execution is to be resumed later

3rd argument:

the address of the result list

results -

NONE

error conditions -

NONE

discussion -

this routine will actually resume execution of the target processes, and optionally modify the address within the target process at which execution is to be resumed; it will use the LM breaker routine for evaluating the address specified at which to resume execution.

procedure name - xcexecute 21am

procedure function (brief) -

tc execute a single instruction on behalf of the target process

arguments -

1st argument

ar indication of the type of instruction to be executed as follows:

FALSE - execute the instruction literally specified by the user

cmlhigh - execute the high level language instruction specified

cmlmachine - execute the machine level instruction specified

2nd argument

if the 1st argument is not FALSE, then this argument specifies whether the user specified an instruction to be executed or the address of an instruction in the target process to be executed

3rd argument

this is the address of a string that either contains the instruction the user specified or the address of an instruction to be executed

4th argument:

the address of the result list

results -

NONE

error conditions -

NONE

discussion -

this routine will execute either an instruction specified by the user or an instruction at the address specified by the user; it will actually use the LM inexec routine for performing its function

UTILITY ROUTINES 22

Many of the routines called from the CLI make use of the following two utility routines for transmitting strings to the CLI to be displayed to the user. 22a

procedure name - prtstr 22b

procedure function (brief) -

to display a string on the user's terminal and/or to write the string in the current output file

arguments -

1st argument:

the address of the string to be displayed and/or written

results -

NONE

error conditions -

NONE

discussion -

The first thing this routine does is to append a carriage-return, linefeed sequence to the end of the passed string.

Then, if output is currently being displayed on the user's terminal, then this routine will call the WRIT-SEQ procedure in the CLI to display the passed string to the user; if output is being written in an output file, then this procedure will (next) write the passed string in the output file.

procedure name - pasnstr 22c

procedure function (brief) -

to display a string on the user's terminal and/or to write the string in the current output file, and to obtain a new value from the user (and optionally new current input mode parameters)

arguments -

1st argument:

the address of the string to be displayed and/or written

2nd argument:

the address of a string to get written with the new value string specified by the user

results -

1st result:

FALSE or a value representing the main input mode keyword specified by the user

2nd result:

a value representing the secondary input mode keyword specified by the user if the main output mode has a secondary mode; FALSE otherwise

error conditions -

NONE

discussion -

The purpose of this routine is to display a string to the user which represents the displaying of some entity, e.g. a cell, in the target process, and then to obtain from the user a new value string for this entity. The user may specify a new current input mode to be used to interpret the new value string.

The first thing this routine does is to append several spaces, followed by a LARROWCHAR, followed by a space, to the passed string.

Then, if output is being sent to the user's terminal, this routine will generate a HELP to the CLI. This HELP will cause the string to be displayed and then the execution of a CML rule that will enable the user to (optionally specify a new current input mode and to) specify a new value to replace the value just displayed. The new value specified by the user, which may be a null string, will be written in the string whose address was passed as the second argument.

Then, if output is being written to an output file, the passed string, followed by the same appended characters mentioned above, followed by the new value string, followed by a carriage-return and a linefeed, will be written on the output file.

Finally, this routine will return; returning the values of the keywords specified by the user for the new input mode (these values will be FALSE if the user did not specify them).

PROGRAMMERS' GUIDE TO LANGUAGE MODULES 23

A language module (LM) is that module in the debugger that is responsible for any language specific function, e.g. interpreting symbolic input according to the semantical and syntactical rules of the current high level language, or displaying a cell in the current high level language. Each language module in the debugger is designed to be run under a specific operating system, and to provide support for one and only one language. For example, there is a separate BCPL language module for support of BCPL on a TENEX and for support of BCPL on an ELF. 23a

The debugger was designed so that a LM can be loaded dynamically by the debugger dispatcher (DD) in response to certain commands by a user. The current practise is to configure a separate debugger SAV file for each language and to make the LM a static module in the debugger. For example, there is a debugger SAV file for the L10 language and a separate debugger SAV file for the JOVIAL language. Each LM is responsible for providing a number of routines (with well defined interfaces) and has available to it a number of routines and data structures in the DD and operating system (OS) modules (also with well defined interfaces). 23b

GROSS STRUCTURE OF A LANGUAGE MODULE 24

A language module consists basically of a dispatch table, routines and data structures that will be called and referenced by other modules of the debugger (hereafter referred to as external routines and data structures), and any routines and data structures (hereafter referred to as support routines and data structures) needed for the support of the external routines and data structures. 24a

At the heart of any LM is its dispatch table. The dispatch table contains: 24b

addresses of external routines, and

addresses of external data structures, and

in some instances, a dispatch table entry is itself an external data structure. (A dispatch table entry that is itself a data structure will be called a simple data structure.)

When the debugger dispatcher receives a language specific request *from the debugger frontend (in response to some user action)*, the DD looks in the LM dispatch table for the address of the LM routine that supports the requested function. The DD will then call the LM routine and expect the LM routine to perform its function and optionally to return some results. If a function is not supported by a language module, then the appropriate entry in the dispatch table shall be 0. 24c

To perform its function, a language module routine may find it necessary to call routines provided by the DD and/or the OS modules, or to reference data structures in these other modules. To do so, the LM routine will use the dispatch table for these other modules and can thus call or reference routines and/or data structures that it does not provide itself. 24c

GENERATING A LANGUAGE MODULE 25

The following discussion is specific for generating a language module designed to run under TENEX. However, the principles involved are the same regardless of what operating system the language module will be run under. 25a

The language module dispatch table MUST be the first thing in each language module. 25b

The sequence of events to generate a LM is (TENEX EXEC commands are in upper case): 25c

Using NLS, edit the file CONFIG.NLS to insure that the switches have the correct value

at the TENEX EXEC, run the program RUNFIL, specifying as the input file the file LOADFE.RUN -- this will properly compile and load the debugger FE

at the TENEX EXEC, run the program RUNFIL, specifying as the input file the file LOADLDR.RUN -- this will properly compile and generate the debugger loader

at the TENEX EXEC, run the program RUNFIL, specifying as the input file the file LOADYL10.RUN or LOADJVL.RUN -- this will properly compile and generate the debugger L10 or JOVIAL LM module appropriately

THE LM DISPATCH TABLE

The symbolic offset names for the entries in the language module dispatch table are contained in the file <nsw-debugger>landsp.nls. Also, the debugger loader contains these definitions. (Note that an offset of 0 refers to the first entry in the dispatch table.)

decimal offset	symbolic offset name	meaning
0	lnini	address of initialization procedure
1	lnsymp	symbol table pointer for this module
2	lntowlm	address of procedure to call when temporarily done with this LM
3	lnrnt	address of procedure to call to resume this LM
4	lntinit	address of procedure to call to initialize tool
5	lntowtl	address of procedure to call when temporarily done with tool
6	lntrntr	address of procedure to call to resume tool
7	lnbpte	address of procedure to call when a breakpoint is hit
8	lnsacr	address of coroutine for displaying and assigning to address ranges
9	lnfmem	address of coroutine for displaying and assigning to content searches
10	lnmass	address of procedure for setting search mask
11	lnmems	address of procedure to execute "Memory Set" command

12	lrsymc	address of procedure to define symbol table
13	lrsymc	address of co-routine to display sym tab
14	lntsta	address of co-routine to give LM status
15	lrnacr	address of procedure to generate new address list for linefeed, uparrow, etc.
16	lrevbadr	address of procedure to evaluate breakpoint address
17	lrnbacr	address of procedure to append breakpoint address to string
18	lrecacr	address of procedure to evaluate resume address
19	lrexec	address of procedure for "Execute" command
20	lrdntl	address of procedure when done with tool
21	lrpdwlm	address of procedure when done with LM
22	lradrstr	address of procedure to convert an address to a string
23	lrcefine	address of co-routine for "Define" command to convert an address to a string
24	lrich	LM current IDB

DETAILED DISCUSSION OF EACH ENTRY IN THE LANGUAGE MODULE DISPATCH TABLE 27

This section will discuss in detail each entry in a language module's dispatch table. Each entry will be discussed under its symbolic offset name. 27a

lmini 27b

entry type - procedure address

procedure function (brief) -

perform language and/or module initialization

when called -

This procedure is called (once and once only) after the language module has been loaded by the debugger. (This is guaranteed to occur after the OS module has been loaded and initialized.)

arguments -

1st argument:

The address of the debugger dispatch table.

2nd argument:

The address of the permanent output mode record.

3rd argument:

The address of the permanent input mode record.

4th argument:

The address of an output string to be used for potential error conditions or for presenting an initialization message to the user.

results -

1st result:

If initialization is successful then this procedure should return TRUE as its first result; if initialization is not successful, then this procedure should return FALSE as its first result. In either case, this procedure may write the output string (whose address is passed as the 4'th argument) with a message to be presented to the user.

error conditions -

Any error conditions detected by this procedure should either be dealt with by this procedure or translated into a FALSE return with an appropriate error message written in the output string.

external data structures maintained -

may wish to modify the DD GFS

discussion -

The function of this procedure is to perform any language and/or module initialization required by the language module.

This procedure should set up the permanent input and output mode records with the defaults for this language. Specifically, the fields CHLANG and CCLANG should be set up in the output mode record; and the fields IHLANG and ICLANG should be set up in the input mode record. In addition any other fields (such as the default radix to be used) may be setup.

An example of language specific initialization might be:

The default debugger character for the address range delimeter character (COMMACHAR) is a comma (','); however, a comma has syntactical and semantical meaning for L10 (and indeed for many languages), therefore a language module initialization procedure might wish to redefine the character that will be used as an address range delimeter at this time (in fact, the L10 module changes a comma to a colon (':') at this time and also changes some other character definitions).

(For a detailed discussion of how to change generic characters see above.)

This procedure may also wish to copy entries from the DD's dispatch table into local variables to speed up future references. This is not necessary, but merely an efficiency consideration, as the address of the DD dispatch table, and its entries (with the exception of those entries that are simple data data structures), is guaranteed not to change for the lifetime of an instance of an LM.

(Note: the addresses of the permanent input and output mode records passed as arguments should not be "remembered" as the passed addresses are merely instances used for LM initialization purposes. Any external routines that reference these records (or the current (as opposed to permanent) input/output mode records) will be passed addresses for the pertinent records (which will probably be a separate instance.)

lrsymp 27c

entry type - symbol table pointer

discussion -

This entry is a symbol table pointer for the symbol table for the LM. (For most languages running on a TENEX this consists of the lefthalf of the word being a negative count of the number of words in the symbol table and the righthalf of the word being the address of the first word of the symbol table.) This entry is not used by the debugger, but is merely a convenience to aid in the debugging of the LM itself.

(Note that the LM symbol table must reside in the same part of the debugger address space allocated to the LM.)

IntoLM 27d

entry type - procedure address

procedure function (brief) -

Perform any cleanup necessary when we are temporarily through with this LM, we are about to load another LM.

when called -

This procedure is called when the user wishes to switch to a different LM and will resume this LM subsequently.

arguments -

1st argument:

the address of a module record (see definition of this record in DDTDEF). Basically a module record is a mechanism within which a module can store information while it is mapped out of the debugger.

results -

1st result:

a boolean that is TRUE if this procedure is successful and FALSE if this procedure is not successful.

error conditions

Should handle any errors itself.

external data structures maintained - NONE

discussion -

This routine should do any cleanup necessary prior to this LM being mapped out. If there are variables whose state the LM wishes to remember while it is mapped out, it may assign debugger global storage, write in this assigned storage, and place the address of this assigned storage in the appropriate

field in the module record whose address it was passed. This module record will be made available to the LM when it is called at its resume use entry.

lrnt 27e

entry type - procedure address

procedure function (brief) -

Perform any reinitialization upon resuming use of an LM.

when called -

This procedure will be called upon resuming use of an LM that has been mapped out while another LM was in use

arguments -

1st argument:

the address of the DD dispatch table

2nd argument:

the address of string for error message

3rd argument:

the address of the module record for the current LM

results -

a boolean -- TRUE if success; FALSE if failure

error conditions

Should handle any errors itself; may return an error message in the passed string.

external data structures maintained - NONE

discussion -

This procedure should reinitialize the LM after it has been mapped back in (after being mapped out for a while so that another LM could be used). It will be passed the address of the same module record that was passed to the ostowasm routine (see above).

lrtinit 27f

entry type - procedure address

procedure function (brief) -

Prepare the LM to debug a new tool.

when called -

This procedure is called the first time a tool is specified for debugging.

arguments -

1st argument:

the address of a tool record

results -

1st result:

a boolean that is TRUE if this procedure is successful and FALSE if this procedure is not successful.

error conditions - NONE

external data structures maintained - NONE

discussion -

This procedure is responsible for setting up any data structures needed for the debugging of a tool. It is called when the debugger first learns of a tool to be debugged. It may want to perform the following functions: copy the symbol

table of the tool, get the run-time state of the tool and build a data structure for the top stack frame of the tool.

lntcwtl 27g

entry type - procedure address

procedure function (brief) -

Save information to allow resumption of debugging of a tool at a later time.

when called -

This procedure is called when a tool is being left temporarily but will be resumed at a later time.

arguments -

1st argument:

the address of the tool record for the tool being left

results -

1st results:

a boolean that is TRUE if this procedure is successful and FALSE if this procedure is not successful.

error conditions - NONE

external data structures maintained - NONE

discussion -

This procedure will be called whenever the debugger is temporarily leaving debug mode for a specific process. It must insure that the process which is being left is in such a state that its execution can be resumed later.

lrtrntr 27h

entry type - procedure address

procedure function (brief) -

Prepare the LM to resume debugging a tool.

when called -

This procedure is called to resume debugging a tool when this is not the first time this tool has been specified in a Debug command.

arguments -

1st argument:

the address of the tool record for the tool being resumed

results -

1st result:

a boolean that is TRUE if this procedure is successful and FALSE if this procedure is not successful.

error conditions - NONE

external data structures maintained - NONE

discussion -

This procedure sets up data structures in the LM so that the tool whose tool record is passed becomes the current tool.

lrtate 271

entry type - procedure address

procedure function (brief) -

Perform any language and/or module specific action required at breakpoint hit (and other, see below) time(s).

when called -

This procedure will be called:

when a target process is specified, and

when a breakpoint is encountered in the target process, and

when a tracepoint is encountered in the target process.

In all instances, this procedure will not be called until after the CS module enter breakpoint procedure has been called.

arguments -

1st argument:

a value (n) that indicates why the procedure is being called this time as follows:

n = 0: user has just specified a target process

n > 0: n is the number of the breakpoint just encountered

n < 0: -n is the number of the tracepoint just encountered

results - NONE

error conditions - NONE

external data structures maintained - NONE

discussion -

The function of this procedure is to obtain the language state of the target process and to build any support data structures that may be required to reflect this state.

For example, when this procedure is called after a target process has been specified, it will probably want to obtain the symbol table for the target process.

Also for example, in the LM for a stack oriented procedural language such as LIC or ALGOL, this procedure might wish to build support data structures to represent the current state of the stack and the current frame (perhaps including the name of the routine that was interrupted).

This procedure will be called after a breakpoint or tracepoint is encountered in the target process or when a new (or the first) target process is specified by the user. It will be called after the OS module enter breakpoint routine has been called and it therefore will have available to it the OS state of the target process (e.g. the registers of the target process).

lrsacr 271

entry type - coroutine address

coroutine function (brief) -

The function of this coroutine is to build strings (using the current output mode) for the display of, and optionally to assign to (using the current input mode), the address ranges specified by the user.

when called -

This coroutine will be called in response to user requests to display and/or assign to address lists.

arguments -

at openport time -

1st argument:

a boolean that is TRUE if this instance of this coroutine should perform assignments to, as well as displaying, address ranges; the boolean will be FALSE if this instance is to be used only for the display of address ranges.

at cycle start time -

1st argument:

address variable -- the address of a starting ARE string

2nd argument:

address variable -- the address of the corresponding ending ARE string

3rd argument:

the address of the output string that should be written by this coroutine

4th argument:

the address of a 2 word block that contains in the first word the address of the current output mode record, and in the second word the address of the current input mode record

at pulse after a positive return -

NONE

at pulse after a negative return -

1st argument:

FALSE, meaning no new value specified by the user; or the address of a string which must be evaluated according the current input mode record passed as the second argument (note that this may be a NULL string which for some cases is different that not specifying a new value string). The value of this string should then be assigned to the previously displayed entity.

2nd argument:

The address of the current input mode record to be used to evaluate the 1st argument.

results -

at openport time -

NONE

all other times -

returns > 0 means it has written the output string and that string should be presented to the user. it is not finished and expects to be called again with no arguments.

returns = 0 means it is done, i.e. it has completed the current cycle; the output string may be NULL or if non null, then it has detected an error and the output string is the error condition to be presented to the user. in either case it may be called again, but it must be presented fresh arguments.

(Note that if this error return occurs on a "first" pulse, it more than likely means that this coroutine got invalid or unsupported address range elements.)

returns < 0 means it has written the output string and that string should be presented to the user and that the user should be asked for a new value to replace the old value just presented to him/her. In this case it expects to be called again with the new value string and new input mode record.

error conditions -

any error conditions detected by this coroutine should be translated into a 0 return with an appropriate error message written in the passed output string. The coroutine should then be ready to accept new arguments to start a new cycle.

external data structures maintained -

may wish to update LSTADIS, LSTVDIS, and LSTEADR depending on the specific operation of this instance (discussed below)

discussion -

This is the main routine for displaying, and modifying, the address space and state information of a target process. This routine is invoked by the DD in response to a number of commands by the user to display and/or assign to address lists.

When this coroutine is OPENPORTed, it will be passed a boolean to indicate if this instance is to be used for displaying and assigning to address ranges, or only for the display of address ranges.

A (complete) cycle consists of the display of, and optionally the assignment to, all the referenced cells in the passed address range. After each cycle, this coroutine must be ready to accept new arguments and to start a new cycle. This coroutine indicates cycle completion by giving a 0 return, with an optional error message written in the output string.

This routine is designed to present a "line" of information at a time to the user. Thus this routine should format the passed output string with a line of information and then PCALL appropriately its caller. In the case of merely displaying address ranges, the formatted output string will have a

carriage-return, linefeed sequence appended to it before it is displayed to the user. In the case of an assignment, the formatted output string will have a space followed by the LARROWCHAR (by default a left arrow (^_)) followed by several spaces appended to it before being presented to the user.

The output line should be formatted according to the current output mode record.

The following guidelines should be adhered to in normal cases:

The DD will display a header line that indicates the address range currently being displayed (and optionally being assigned to).

For the display of data structures, the first line output should be a header that indicates the type of the data structure and the address (discussed below) of the instance of this data structure (are optionally other data structure dependent information, e.g. the length of a record).

Successive lines of output should contain either the entire data structure (e.g. an entire string) or successive elements of the data structure (e.g. successive fields of a record).

(It is recommended that these successive lines have 3 spaces at their front.)

For the display of certain gross address range types (e.g. stack frames), the first output line should be a header indicating the type of gross address range and any other pertinent header type information.

Successive lines of output should contain further information relevant to this address range.

(It is recommended that these successive lines have 3 spaces at their front.)

For the normal display of cells in the address space of the target process the output line should be formatted as follows:

the address of the current cell being displayed.

addresses should be displayed either numerically in the current output mode radix, or as a symbol optionally followed by a plus sign and a numeric offset (in the current output mode radix), depending on the current output mode record field OSYMADR. Even if OSYMADR indicates that addresses should be displayed as a symbol plus an offset, if the offset is greater than MAXOFFSET (in the DD dispatch table), then the address should be displayed numerically.

the address should be followed by a slash character ('/'), followed by 3 spaces.

and finally the value of the cell(s) according to the current output mode record.

The following guidelines should be adhered to for exceptional cases:

If the current output mode specifies a data structure not supported for the current language, e.g. output as lists for COBOL, then this coroutine ought to give an 0 return with an appropriate message.

If the current output mode specifies a data mode not yet implemented for the current language, e.g. output floating point numbers for L10, then this coroutine ought to give an 0 return with an appropriate message.

If the current output mode specifies a data mode not yet implemented for the current language, e.g. output in source language for L10, then this coroutine may substitute a subset output mode for this case, e.g. output in assembly language.

If the combination of address range gross type and current output mode conflict, e.g. an output mode of memdump (tell where symbols are defined) and a gross type of dfra (stack frame), then the gross type of address range should take precedence and this coroutine should format the output string in the manner appropriate to the gross type.

If this instance of the coroutine is an assignment instance, and the address range specifies a type for which assignment is not meaningful (e.g. signal status),

then this instance should be treated as only a display instance for the current cycle.

Note that for some address ranges it may be meaningful to assign to part of the address range and only to display other parts of the address range. Remember that this coroutine can control what will and will not be assigned to by the type of PCALL back to its owner that it performs (greater than 0 for display only; or less than 0 for display and assign).

This coroutine is responsible for maintaining the following DD data structures according to the following guidelines:

LSTEADR - after this coroutine evaluates an ARE it should write the resulting value in lsteadr.

(Note: it is the discretion of the LM to decide whether or not to do this for certain gross address ranges, e.g. the LM may wish to update LSTEADR for a normal memory range address range but not for a signal status address range.)

LSTADIS - just prior to displaying a line of information, this coroutine should update the data structure LSTADIS (using the appropriate DD external routine) with the value of the address about to be displayed.

(Once again it is the discretion of the LM to decide whether or not to do this for certain gross address ranges, e.g. the LM may wish to update LSTADIS for a normal memory range address range but not for a signal status address range.)

LSTVDIS - just prior to displaying a line of information, this coroutine should update the data structure LSTVDIS with the value of the cell about to be displayed.

(Once again it is the discretion of the LM to decide whether or not to do this for certain gross address ranges or certain data structure types, e.g. the LM may not wish to update LSTVDIS for displaying of memory as lists.)

trimem 27k

entry type - coroutine address

coroutine function (brief) -

The function of this coroutine is to build strings (using the current output mode) for the display of, and optionally to assign to (using the current input mode), cells within the specified address range that meet certain specified content requirements.

when called -

This coroutine will be called in response to the user issuing the FIND command.

arguments -

at openport time -

1st argument:

a boolean that is TRUE if this instance of this coroutine should perform assignments to, as well as displaying, filtered address ranges; the boolean will be FALSE if this instance is to be used only for the display of address ranges.

at cycle start time -

1st argument:

the address of a 3 word parameter block that contains the following:

1st word: a value indicating what kind of search to perform as follows:

cmlreferences:

search for cells whose address portion is equal to the passed searchee value in the passed

address range (when both have been appropriately masked).

crlcontent:

search for cells in the passed address range whose content is equal to the passed searchee value (when both have been appropriately masked).

crlnot:

search for cells in the passed address range whose content is not equal to the passed searchee value (when both have been appropriately masked).

2nd word: FALSE indicating that no mask should be used in performing the search; or the address of a string that is to be evaluated to become the mask to use. If this string is a NULL string, then use the default debugger mask (DEFMASK).

Even if this word is FALSE implying no mask, if this is a reference search an implicit mask may be used for some environments (e.g. under TENEX a reference search actually uses a mask that only causes the right half of words to be examined).

3rd word: the address of the searchee string that must be evaluated to the searchee value according to the current input mode record.

2rd argument:

the address of a 2 word data block containing the following:

1st word: the address of the first ARE string of an address range

2nd word: the address of the corresponding second ARE string of the above address range

3rd argument:

the address of a 2 word block that contains in the first

word the address of the current output mode record, and in the second word the address of the current input mode record

4th argument:

the address of the output string that should be written by this coroutine

at pulse after a positive return -

NONE

at pulse after a negative return -

1st argument:

FALSE, meaning no new value specified by the user; or the address of a string which must be evaluated according the current input mode record passed as the second argument (note that this may be a NULL string which for some cases is different from not specifying a new value string). The value of this string should then be assigned to the previously displayed entity.

2nd argument:

The address of the current input mode record to be used to evaluate the 1st argument.

results -

see lrsadr above

error conditions -

any error conditions detected by this coroutine should be translated into a 0 return with an appropriate error message written in the passed output string. The coroutine should then be ready to accept new arguments to start a new cycle.

external data structures maintained -

LSTADIS, LSTVDIS, and LSTEADR

discussion -

This routine is used for displaying, and optionally assigning to, cells within an address range that meet certain content requirements. This routine is called by the DD in response to the FIND command issued by the user.

When this coroutine is OPENPORTed, it will be passed a boolean to indicate if this instance is to be used for displaying and assigning to the cells in the passed address range that meet the specified content requirements, or only for the display of those cells.

The operation of this routine is identical to the operation of lrsacr (see above) and it should obey all the same conventions (including the maintenance of the debugger wide data structures). The only difference is that before a cell is considered for display (and optional modification), it must pass the specified content requirements as follows:

Both the potential cell to be displayed and the passed searchee value are masked (logically anded) with the appropriate mask.

If the user does not specify to use a mask (indicated by word 2 of the 1st argument being FALSE), then the mask to be used is one that will select an entire word for cmlcontent or cmlnot searches; or one that selects the address portion of a cell for cmlreferences searches.

If the user does specify a mask that is a non null string, then the mask to be used is the evaluation of this string according to the current input mode. This mask is used regardless of the type of search.

If the user specifies a null string as a mask, then the mask to be used is the external data structure DEFMASK in the DD, once again regardless of the type of search.

The resulting 2 masked values are then compared and if equal and if this is a cmlcontent search or a cmlreferences search, then this cell is considered to pass the filter and should be displayed to the user (and optionally modified).

If the resulting 2 values are unequal, then this cell is considered to pass only if this is a cmlnot search.

lrmass 271

entry type - procedure address

procedure function (brief) -

tc set the DD external data structure DEFMASK. the debugger default mask for searches and memory setting

when called -

This procedure is called by the DD in response to the user command to set the default mask.

arguments -

1st argument:

FALSE or the address of a (possibly NULL) string to be evaluated to become the debugger default mask

2nd argument:

the address of the current input mode record to use in the evaluation of the 1st argument

3rd argument:

the address of an output string for possible (error) messages

results -

1st result: TRUE indicating success; FALSE indicating error detected.

in either case, if the output string is non NULL (which it should be for a FALSE return), it will be presented to the user.

error conditions -

any error conditions detected by this procedure should be

translated into a 0 return with an appropriate error message written in the output string.

external data structures maintained -

DEFMASK assuming all goes well (discussed below)

discussion -

The function of this procedure is to set the debugger default mask (a DD external simple data structure whose offset in the DD's dispatch table is dodfmk) which is used by the FIND and MEMORY SET commands.

The passed mask string should be evaluated according to the passed current input mode record and the resulting value written in DEFMASK.

Any error conditions, such as no mask string, a null mask string, or an invalid mask string, should generate a 0 return with an appropriate error message written in the output string.

This procedure should not modify any other external data structures.

lnmems 27m

entry type - coroutine address

coroutine function (brief) -

to set all cells (masked appropriately) in the specified address range to the specified new value (masked appropriately).

when called -

This coroutine is invoked by the DD in response to a "MEMORY SET" command issued by the user.

arguments -

at OPENPORT time -

1st argument:

FALSE or the address of a (possibly null) new value string.

2nd argument:

the address of the current input mode record to use for the evaluation of the new value string and for the evaluation of any mask specified.

3rd argument:

the address of the current output mode record to use for any output strings written

4th argument:

TRUE to indicate the use of the mask as specified by the next argument; FALSE means don't use a mask.

5th argument:

FALSE to mean use the debugger default mask as the mask to use; or the address of a (possibly null) string to be evaluated, according to the current input mode record, as the mask to use.

6th argument:

the address of an output string for possible (error) messages.

at cycle start time -

1st argument:

the address of a starting ARE string

2nd argument:

the address of the corresponding second ARE string

3rd argument:

the address of an output string

all other times -

NONE

results -

at OPENPORT time -

this coroutine should return a boolean in its EXIT PCALL phrase that is TRUE indicating that the passed arguments have been evaluated successfully, or FALSE if the passed arguments could not be evaluated successfully. In the case of a FALSE return the output string should contain an error message.

all other times -

returns > 0 means it has written the output string and that string should be presented to the user. It is not finished and expects to be called again with no arguments.

returns = 0 means it is done, i.e. it has completed the current cycle; the output string may be NULL or if non null, then it has detected an error and the output string is the error condition to be presented to the user. In either case it may be called again, but it must be presented fresh arguments.

error conditions -

any error conditions detected by this routine should be translated into the appropriate 0 (or FALSE) return (at OPENPORT or other times) with an appropriate error message in the output string

external data structures maintained -

LSTEADR

discussion -

This routine is used to set all cells in a specified address range to a specific value. An optional mask may be employed to select only certain fields of the cells.

If this routine completes a cycle successfully, it should generate a positive return indicating the number of words set

(by writing the output string), and then give a cycle terminating PCALL back to its owner.

If this routine encounters a problem during a cycle, it should generate a cycle terminating PCALL with an appropriate error message in the output string. The error message should include some indication of how many, and the addresses, of the cells that were set, as well as some indication of why it could not complete its cycle completely.

lnsymg 27n

entry type - procedure address

procedure function (brief) -

Get a copy of the symbol table for this tool into local debugger core.

when called -

This procedure is called when the user issues a Debug command or Symbol Table Pointer command.

arguments -

1st argument:

the address of the tool record

2nd argument:

the address in the tool of a symbol table pointer or zero to get the most recent symbol table accessed for this tool

3rd argument:

the address in the tool of a symbol table pointer for the undefined symbol table or zero to get the most recent undefined symbol table accessed for this tool

4th argument:

the address of a string for an error message

results - NONE

error conditions - NONE

external data structures maintained - NONE

discussion -

If this is the first time a symbol table has been accessed for this tool, this procedure will generate a file name and create a scratch file for the symbol table.

Each time this procedure is called, it will either read the symbol table in from the target process, or from a file containing the symbol table. In the case that the symbol table is read in from the target process, this procedure will write the symbol table out to the symbol table file.

lrsymc 27o

entry type - coroutine address

coroutine function (brief) -

The function of this coroutine is to build strings (using the current output mode) that show the contents of the symbol table. In VERBOSE mode, in addition to the names and extents of the symbol table blocks, it will show all the symbols in the table.

when called -

This coroutine will be called in response to user issuing a Symbol Table Display command.

arguments -

at openport time -

1st argument:

the address of the current output mode record

2nd argument:

a boolean that is TRUE for verbose mode and FALSE for brief mode

3rd argument:

the address of a string with a block name or zero to display information on all symbol table blocks

at cycle start time -

1st argument:

the address of the tool record

2nd argument:

the address of a string for display string

at pulse after a positive return -

NONE

at pulse after a negative return -

same as at cycle start time

results -

at openport time - NONE

all other times -

1st result:

a boolean that is TRUE if there is more information to display or FALSE if all requested information has been shown

error conditions - NONE

external data structures maintained - NONE

discussion

this coroutine is used to interpret a symbol table for the user.

lrtsta 27p

entry type - coroutine address

coroutine function (brief) -

Build strings to show the user the language state of a process.

when called -

This procedure is called in response to a user typing a Status command

arguments -

at openport time -

1st argument:

the address of the current output mode record

2nd argument:

a boolean that is TRUE for verbose mode and FALSE for brief mode

at cycle start time -

1st argument:

the address of the tool record

2nd argument:

the address of a string for display string

3rd argument:

an integer that is the number of blanks to precede the

display string: used for indenting various parts of the displayed information

at pulse after a positive return -

1st argument:

the address of a string for display string

at pulse after a negative return -

same as at cycle start time

results - NONE

at openport time - NONE

all other times -

1st result:

a boolean that is TRUE if there is more information to display or FALSE if all requested information has been shown

error conditions - NONE

external data structures maintained - NONE

discussion

this routine is used to obtain and format for the user, the language state of a process.

lrracr 27o

entry type - procedure address

procedure function (brief) -

Generate a new address list for the linefeed, uparrow, tab and poundsign commands.

when called -

Called in response to the user typing a linefeed, uparrow, tab or poundsign command.

arguments -

1st argument:

An integer whose value indicates the character typed by the user. The following are the valid values:

cmllf -- linefeed

cmluparrow -- uparrow

cmltab -- tab

cmlpound -- poundsign

These value are defined in the Fronterd interface module

2nd argument

the address of the string with the last starting address element

3rd argument

the address of the string with the last ending address element

4th argument:

the address of a string with the dispatchers idea of new access list

5th argument:

the address of a string for the new access list generated by this procedure

6th argument:

the address of the output record from the last command

7th argument:

the address of the input record for the last command

results -

1st result:

a boolean that is TRUE if this procedure generates a new access list or FALSE if the dispatcher's idea of the new access list is correct

error conditions - NONE

external data structures maintained - NONE

discussion -

whether or not this procedure generates a new address list depends upon the type of request and the type of the last access list given. In some cases the dispatcher can determine the new address list while in other cases the new address list is language and/or operating system dependent and consequently must be supplied by this procedure. For example, the dispatcher can generate a new address list if the last access list was a single cell and the command is a linefeed, but this procedure must generate the new address list if the last access list an entry in a JOVIAL table.

trevbacr 27r

entry type - procedure address

procedure function (brief) -

To evaluate a symbolic address (as specified by a user) to be used as a breakpoint address.

when called -

this routine is called whenever any debugger routine needs to evaluate an address supplied by the user to used as the address of a breakpoint.

arguments -

1st argument

the address of a string containing the symbolic address supplied by the user

2nd argument

the address of the current input mode record to be used to evaluate argument 1

3rd argument

the address of a string to get any error messages

results

1st result

TRUE if the supplied address could be evaluated successfully; FALSE otherwise, with an appropriate message written in the provided error string

2nd result

if the 1st result was TRUE, then this result should be the

value that the passed argument evaluated to; if the 1st result was FALSE, then this result is not used

error conditions

any errors detected by this routine should be reflected by a FALSE return with an appropriate error message for the user

external data structures maintained - NONE

discussion -

This routine should evaluate the passed symbolic address according to the current input mode record and return the result. This routine, if desired, may in fact return a value that does not correspond exactly to the passed argument; e.g., if the passed argument evaluates to the start of a procedure, and if the current language has some instructions at procedure entry that must be executed before it is really in the procedure, then this routine may wish to return as a value the address of the first instruction after the procedure initialization code.

lrrbacr 27s

entry type - procedure address

procedure function (brief) -

to append the address at which a breakpoint is set to the passed string

when called -

whenever the debugger is presenting information about breakpoints to the user

arguments

1st argument

the address of the appropriate Debugger Instance Record (DIREC)

3rd argument

the address of the string to be appended to

3rd argument

the address of the current output mode record to be used for formatting any results

results

should return TRUE after appending to the passed string

error conditions - NONE

external data structures maintained - NONE

discussion -

This routine should interpret the appropriate fields of the passed DIREC and append to the passed string an indication of which breakpoint, etc., and at which address, this instance record refers to.

trecadr 27t

entry type - procedure address

procedure function (brief) -

To evaluate a symbolic address (as specified by a user) to be used as a continuation address.

when called -

this routine is called whenever any debugger routine needs to evaluate an address supplied by the user to used as a continuation address, i.e., a new value for the process' pc.

arguments -

1st argument

the address of a string containing the symbolic address supplied by the user

Src argument

the address of the current input mode record to be used to evaluate argument 1

Trc argument

the address of a string to get any error messages

results

1st result

TRUE if the supplied address could be evaluated successfully; FALSE otherwise, with an appropriate message written in the provided error string

2nd result

if the 1st result was TRUE, then this result should be the value that the passed argument evaluated to; if the 1st result was FALSE, then this result is not used

error conditions

any errors detected by this routine should be reflected by a FALSE return with an appropriate error message for the user

external data structures maintained - NONE

discussion -

This routine should evaluate the passed symbolic address according to the current input mode record and return the result. The value returned by this routine (on a successful return) will be used to set the pc of the target process.

[rexec 270

entry type - procedure address

procedure function (brief) -

tc compile, assemble, or copy code into the target process address space for subsequent execution

when called -

in response to the EXECUTE command by a user

arguments -

1st argument

the address of the current input mode record

2nd argument

if TRUE, the next argument represents the address of code in the target process' address space to be executed; if FALSE, the next argument is a string to be compiled/assembled into the target process' address space

3rd argument

see 2nd argument

4th argument

the address in the target process at which to place any compiled/assembled code

5th argument

the address of a string to get any error messages

results

1st result

FALSE, or the next free address in the target process' address space that can be used for subsequent calls on this routine

error conditions

any errors detected by this routine should generate a FALSE return with an appropriate error message in the provided string

external data structures maintained - NONE

discussion -

This routine will compile/assemble/copy code into the target process' address space for subsequent execution. This routine provides the support for the EXECUTE command.

lrcntl 27v

entry type - procedure address

procedure function (brief) -

To do any cleanup necessary when the debugger is done with a tool.

when called -

Whenever a tool is about to disappear, e.g., when the user gives the DONE command, or when one process kills another process in a process oriented operating system.

arguments -

1st argument

the address of the tool record associated with the process that is about to disappear

results - NONE

error conditions - NONE

external data structures maintained

lcich

discussion -

This routine should clearup any data structures associated with the tool that is about to disappear. When it is done it should zero the lcidh data structure, indicating that this LM does not have a current tool being debugged.

trpcwlm 27w

entry type - procedure address

procedure function (brief) -

To do any cleanup necessary when the debugger is done temporarily with this LM.

when called -

In response to commands by a user that indicate that there is no further need for this LM.

arguments -

1st argument

the address of any Module Record (MR) associated with this module. The MR may have been established previously when this LM was given any opportunity for temporarily being left.

results - NONE

error conditions - NONE

external data structures maintained - NONE

discussion -

This routine should release any storage that it had previously acquired as it (i.e., the LM) will not be used again.

lmgrstr 27x

entry type - procedure address

procedure function (brief) -

to append the name of the passed value to the passed string

when called -

wherever any debugger routine wishes to obtain the symbolic name for a specific value

arguments -

1st argument

the value whose symbolic name is desired

2nd argument

the address of the string to be appended to

3rd argument

the address of the current output mode record.

results

TRUE on success

error conditions - NONE

external data structures maintained - NONE

discussion -

This routine should obtain, and format according to the current output mode record, the symbolic name for the passed address.

It should then append the generated string to the passed string.

lrcefire 27y

entry type - coroutine address

procedure function (brief) -

The function of this coroutine is to get a definition from the user of a language specific data structure in the target process. An example of such a data structure is a JOVIAL table.

when called -

This procedure is called when the user issues a Define command.

arguments -

at compport time - NONE

at cycle start time -

1st argument:

the address of a string containing the first part of the definition of the data structure; the format and contents of the text of the string is completely unrestricted and will be interpreted by the LM

2nd argument:

an address list the specifies the location of the data structure in the target process

3rd argument:

the address of the current input mode record

4th argument:

the address of the string for error messages

at pulse after a positive return -

1st argument:

the address of a string containing further specification of the data structure definition or a zero indicating that the data structure is fully specified; the format and contents of the text of the string is completely unrestricted and will be interpreted by the LM

2nd argument:

the address of the string for error messages

at pulse after a negative return -

the same as at cycle time

results - NONE

a boolean whose value is TRUE if this procedure expects further specification from the user or FALSE if the definition is complete

error conditions -

This procedure finds if there is an error in the definition text.

external data structures maintained -

The data structures that are maintained is language dependent.

trich 27z

entry type - simple data structure. an INTEGER

discussion -

This is the IDH of the current tool for this LM

PROGRAMMERS' GUIDE TO OPERATING SYSTEM MODULES 28

An Operating System Module (OSM) is that module in the debugger that is responsible for all reading and manipulation of the address space and state information of a target process. Each OSM in the debugger is designed to: 28a

be run under a specific operating system, and to

read and/or manipulate the address space and state information of any target processes running running under a (potentially different) system; e.g. there is one OSM for running under a TENEX to manipulate target processes running under ELF.

An OSM is loaded dynamically by the debugger dispatcher (DD) in response to certain commands by a user. Each OSM is responsible for providing a number of routines (with well defined interfaces) and has available to it a number of routines and data structures in the DD and language modules (LM) (also with well defined interfaces). 28b

MOTIVATION FOR AN OPERATING SYSTEM MODULE 29

Any interactive debugger must provide facilities for examining and manipulating the address space and state information of the process it is debugging. The function of the OSM is to isolate all such code into a single module with a well defined interface, and to force routines in the debugger that perform such examinations and manipulations, to use routines provided by the OSM. 29a

The isolation of these functional routines into a single module then makes it possible to dynamically load the module or to replace one module with another. Thus it becomes possible to debug either processes that live on the same machine as the debugger by using one OSM, or to debug processes on a remote machine merely by loading the appropriate OSM. 29b

This is possible, e.g., because a debugger routine that wishes to examine a cell in the address space of a target process always calls a routine in the OSM to return the contents of the specific cell, rather than the debugger routine reading the cell directly. The debugger routine need not know how the OSM obtained the contents of the cell, or for that matter, whether or not the target process is a process on the same machine as the debugger.

How the OSM performs its tasks is of no concern to the rest of the debugger. All that really matters is that the routines in the OSM obey the well defined interface conditions. 29c

Thus, one OSM designed to run under TENEX and the MSG system, and designed to debug processes under the same environment, may perform many of its tasks by sharing pages with the target process;

another OSM designed to run under TENEX and the MSG system, and designed to debug processes running under ELF, may perform its functions by using an "ELF" protocol for target process examination and manipulation;

and yet another OSM designed to run under TENEX and the MSG system, and designed to debug processes running on a YUK-7, may perform its functions by using a "YUK-7" protocol.

Yet, in all these cases, the higher level debugger routines responsible for interpreting the bits of the target process and presenting information to the user, does not need to know how the OSM performed its tasks. 29c

(This is an oversimplification, as indeed the higher level routines must be aware of such things as the word size, etc. of the target process, but the higher level routines do NOT need to know how the bits were obtained.)

GROSS STRUCTURE OF AN OPERATING SYSTEM MODULE 30

An OSM consists basically of a dispatch table, routines and data structures that will be called and referenced by other modules of the debugger (hereafter referred to as external routines and data structures), and any routines and data structures (hereafter referred to as support routines and data structures) needed for the support of the external routines and data structures. 30a

At the heart of any OSM is its dispatch table. The dispatch table contains: 30b

addresses of external routines, and

addresses of external data structures, and

in some instances, a dispatch table entry is itself an external data structure.

When the debugger dispatcher or a language module has a need to read or write the address space or state information of a target process, the DD or LM looks in the OSM dispatch table for the address of the OSM routine that supports the requested function. The DD or LM will then call the OSM routine and expect the OSM routine to perform its function and optionally to return some results. 30c

(All functions discussed in this document (with the exception of the initialization routine) MUST be provided by an OSM.)

(In fact, a LM need only lookup up entities in the DD's dispatch table as the DD will, as part of its initialization sequence prior to loading a LM, copy the relevant entries from the OSM's dispatch table into the DD's dispatch table. Thus, a LM need only know about the DD's dispatch table.)

To perform its function, an operating system module routine may find it necessary to call routines provided by the DD and/or the language modules, or to reference data structures in these other modules. To do so, the OSM routine will use the dispatch table for these other modules and can thus call or reference routines and/or data structures that it does not provide itself. 30d

GENERAL DISCUSSION OF TARGET PROCESS MANIPULATION 31

An OSM may perform its functions in any manner it chooses as long as it obeys the specified interface conditions. The address space allocated to the OSM (under TENEX) is fairly large, and the functions provided by an OSM are fairly simple and should not require much code. It is therefore expected that an OSM will use most of its access space to keep local copies of the address space of the target process in its own address space. Thus, when a higher level debugger routine requests the reading of a cell, the OSM may already have a copy of the cell and may not have to go to the target process to obtain the cell. The management of its own address space is entirely up to the OSM. 31a

(The OSM designed to run under TENEX and to debug processes on the same TENEX, will share pages with the target process. This OSM keeps a local map of what pages it currently has locally.)

An OSM that is designed to debug processes running on a separate machine from the one on which the debugger is running must, of course, get a fresh copy of the target process memory and state information each time a breakpoint or tracepoint is encountered (or when a new target process is specified), since there is no guarantee that its local copy will be valid after the target process has been allowed to execute for any length of time. 31b

The state vector is the mechanism by which the higher level debugger routines examine and modify the state of the target process. There will be one fixed format state vector for each machine that a target process can run on, and this format will be published and known by the LPs and the DD. 31c

Thus there will be one state vector format for any process running on a TENEX;

(The first 16 words of the TENEX state vector consist of the target process' registers. However, since the PDP-10 considers the registers to be part of the address space of an active process, the registers may be read and/or manipulated either by state vector manipulation or by normal access space manipulation and the specification of the appropriate accesses.)

one state vector format for any process running on an ELF;

etc.

GENERATING AN OPERATING SYSTEM MODULE 32

The following discussion is specific for generating an operating system module designed to run under TENEX. However, the principles involved are the same regardless of what operating system the OSM will be run under. 32a

The operating system module dispatch table MUST be the first thing in each operating system module. 32b

The sequence of events to generate a OSM is (TENEX EXEC commands are in upper case): 32c

using NLS, edit the file CONFIG.NLS to insure that the switches have the correct value

at the TENEX EXEC, run the program RUNFIL, specifying as the input file the file LOADFE.RUN -- this will properly compile and load the debugger FE

at the TENEX EXEC, run the program RUNFIL, specifying as the input file the file LOADLDR.RUN -- this will properly compile and generate the debugger loader

at the TENEX EXEC, run the program RUNFIL, specifying as the input file the file LOADTNX.RUN -- this will properly compile and generate the debugger OSM module for running on a TENEX and debugging processes on the same TENEX

THE OSM DISPATCH TABLE

The symbolic offset names for the entries in the operating system module dispatch table are contained in the file <nsw-debugger>dsicsp.mls. Also, the debugger loader contains these definitions. (Note that an offset of 0 refers to the first entry in the dispatch table.)

decimal offset	symbolic offset name	meaning
0	dsini	address of initialization procedure
1	dssymp	symbol table pointer for this module
2	dstdwosm	address of procedure to call when temporarily finished with this OSM
3	dsrnt	address of procedure to call when resuming use of this OSM
4	dstinit	address of procedure to call when first pointed at a tool to be debugged
5	dstdwtl	address of procedure to call when temporarily done debugging a tool
6	dstrrtr	address of procedure to call when resuming debugging a tool
7	dsbpte	address of procedure to call when a breakpoint is hit
8	dsmsta	address of coroutine to perform "memstat" function
9	dsrd1w	address of procedure to read 1 word from the target process' address space
10	dsrdnw	address of procedure to read n words from the target process' address space
11	dsstat	address of coroutine to get target process Operating System status

decimal offset	symbolic offset name	meaning
12	cswr1w	address of procedure to write 1 word in the target process' address space
13	cswrrw	address of procedure to write n words in the target process' address space
14	cssrch	address of procedure to do content searches in the target process' address space
15	csgpfs	address of procedure to get the state vector of the target process
16	csspfs	address of procedure to set the state vector of the target process
17	csalcs	address of procedure to call to get debugger storage
18	csrels	address of procedure to call to free debugger storage
19	csrmbrk	address of procedure to call to remove a breakpoint
20	csinbrk	address of procedure to call to insert a breakpoint
21	csesadr	address of procedure to call to set target process resume address
22	csgpc	address of procedure to call to get target process pc
23	csdntl	address of procedure to call when done debugging a tool
24	csstop	address of procedure to call to stop a tool
25	csgo	address of procedure to call to resume a tool
26	cspdwosm	address of procedure to call when finished with this OSM

decimal offset	symbolic offset name	meaning
27	csesta	address of coroutine to call to get target process last error message
28	csfsta	address of coroutine to call to get list of files opened by the target process
29	csfsav	address of procedure to call to save a target process address space
30	csidh	the IDH of the current target process
31-81	cssvec	state vector for the current target process; cssvec is the symbolic offset for the first word of the state vector

DETAILED DISCUSSION OF EACH ENTRY IN THE OSM'S DISPATCH TABLE 34

This section will discuss in detail each entry in an operating system module's dispatch table. Each entry will be discussed under its symbolic offset name. 34a

csini 34b

entry type - procedure address

procedure function (brief) -

perform module initialization

when called -

This procedure is called (once and once only) after the operating system module has been loaded by the debugger.

arguments -

1st argument:

The address of the debugger dispatch table.

2nd argument:

The address of an output string to be used for potential error conditions or for presenting an initialization message to the user.

results -

1st result:

If initialization is successful then this procedure should return TRUE as its first result; if initialization is not successful, then this procedure should return FALSE as its first result. In either case, this procedure may write the output string (whose address is passed as the 2nd argument) with a message to be presented to the user.

error conditions -

Any error conditions detected by this procedure should either be dealt with by this procedure or translated into a FALSE return with an appropriate error message written in the output string.

discussion -

The function of this procedure is to perform any module initialization required by the operating system module.

This procedure may wish to copy entries from the DD's dispatch table into local variables to speed up future references. This is not necessary, but merely an efficiency consideration, as the address of the DD dispatch table, and its entries (with the exception of those entries that are simple data data structures), is guaranteed not to change for the lifetime of an instance of an OSM.

dssymp 34c

entry type - symbol table pointer

discussion -

This entry is a symbol table pointer for the symbol table for the OSM. (For most languages running on a TENEX this consists of the lefthalf of the word being a negative count of the number of words in the symbol table and the righthalf of the word being the address of the first word of the symbol table.) This entry is not used by the debugger, but is merely a convenience to aid in the debugging of the OSM itself.

(Note that the OSM symbol table must reside in the same part of the debugger address space allocated to the OSM.)

dstdwdsr 34d

entry type - procedure address

procedure function (brief) -

Perform any cleanup necessary when we are temporarily through with this OSM, we are about to load another OSM.

when called -

This procedure is called just prior to its being mapped out in order to make room for another OSM

arguments -

1st argument

the address of a module record (see definition of this record in CDTDEF). Basically a module record is a mechanism within which a module can store information while it is mapped out of the debugger.

results -

Should return TRUE.

error conditions -

Should handle any errors itself.

discussion -

This routine should do any cleanup necessary prior to this OSM being mapped out. If there are variables whose state the OSM wishes to remember while it is mapped out, it may assign debugger global storage, write in this assigned storage, and place the address of this assigned storage in the appropriate field in the module record whose address it was passed. This module record will be made available to the OSM when it is called at its resume use entry.

osrnt 34e

entry type - procedure address

procedure function (brief) -

Perform any reinitialization upon resuming use of an OSM

when called -

This procedure will be called upon resuming use of an OSM that has been mapped out while another OSM was in use

arguments -

1st argument

the address of the DD dispatch table

2nd argument

the address of a string to get any error messages

3rd argument

the address of a module record

results -

Should return TRUE after successful reinitialization

error conditions -

Should handle any errors itself; may return an error message in the passed string.

discussion -

This procedure should reinitialize the OSM after it has been mapped back in (after being mapped out for a while so that another OSM could be used). It will be passed the address of the same module record that was passed to the ostdwosm routine (see above).

ostinit 34f

entry type - procedure address

procedure function (brief) -

To initialize any data structures needed for debugging a process.

when called -

This procedure is called each time the debugger is pointed at a new process.

arguments -

1st argument

the address of a tool record (see DDTDEF).

results -

Should return TRUE after building any needed data structures

error conditions -

should handle any errors internally

discussion -

This procedure is called when the debugger is pointed at a process for the first time. It is passed the address of a tool record (the mechanism by which the debugger maintains its internal knowledge of a process). It should fill in the appropriate fields of this record, as well as updating the osidh data structure.

osdcwtl 34c

entry type - procedure address

procedure function (brief) -

To set the state of a target process so that execution of the process may be continued later.

when called -

This procedure is called either when the user requests to continue execution of all target processes, or when the user points the debugger at another process to become the current target process.

arguments -

1st argument

the address of the tool record that corresponds to the tool we are currently leaving.

results -

NONE

error conditions -

NONE

discussion -

This procedure will be called whenever the debugger is temporarily leaving debug mode for a specific process. It should indicate (by setting the osidh data structure to 0) that there is not a current process being debugged. It must also insure that the process which is being left is in such a state that its execution can be resumed later.

ostrntr 34h

entry type - procedure address

procedure function (brief) -

To resume debugging of a process.

when called -

Whenever the debugger is re-pointed at a process for debugging purposes.

arguments -

1st argument

the address of the tool record for the process about to become the current target process

results -

Should return TRUE upon successfully making the process current

error conditions -

NONE

discussion -

This procedure will be called whenever the debugger is re-pointed at a process. It should update the osidh data structure. For most OSMs, it should remove, but remember, all breakpoints and single step instructions inserted in the target

process address space so that when a user looks at the address space (s)he sees the real code and not code placed there by the debugger.

ostpte 341

entry type - procedure address

procedure function (brief) -

Perform any operating system and/or module specific action required at breakpoint hit (and other, see below) time(s).

when called -

This procedure will be called:

when a target process is specified, and

when a breakpoint is encountered in the target process, and

when a tracepoint is encountered in the target process.

arguments -

1st argument:

a value (n) that indicates why the procedure is being called this time as follows:

n = 0: user has just specified a target process

n > 0: n is the number of the breakpoint just encountered

n < 0: -n is the number of the tracepoint just encountered

results -

NONE

error conditions -

NONE

discussion -

The function of this procedure is to obtain the state vector of the target process and to build any support data structures that may be required to reflect this state.

This procedure may wish to remove from the address space any code that was inserted to implement breakpoints (or tracepoints) and replace such code with the original code.

This procedure will be called after a breakpoint or tracepoint is encountered in the target process or when a new (or the first) target process is specified by the user.

csmsta 34j

entry type - coroutine address

coroutine function (brief) -

The function of this coroutine is to build strings (using the current output mode) for the display of the address space of the target process. (This coroutine implements the TENEX "MEMSTAT" command.)

when called -

This coroutine will be called by a LM in response to a user's request to display the gross address range of type dadr.

arguments -

at openport time -

NONE

at cycle start time -

1st argument:

the address of the output string that should be written by this coroutine

2nd argument:

the address of the current output mode record to be used in the building of the output string

at pulse after a positive return -

NONE

at pulse after a negative return -

NONE

(This coroutine should not generate any negative returns.)

results -

at openport time -

NONE

all other times -

returns > 0 means it has written the output string and that string should be presented to the user. it is not finished and expects to be called again with no arguments.

returns = 0 means it is done, i.e. it has completed the current cycle; the output string may be NULL or if not null, then it has detected an error and the output string is the error condition to be presented to the user. in either case it may be called again, but it must be presented fresh arguments.

(Note that if this error return occurs on a "first" pulse, it more than likely means that this coroutine got invalid or unsupported address range elements.)

error conditions -

any error conditions detected by this coroutine should be translated into a 0 return with an appropriate error message written in the passed output string. The coroutine should then be ready to accept new arguments to start a new cycle.

discussion -

This coroutine is used to build strings (for eventual display to a user) that describe the gross utilization of the access space of the target process.

This coroutine should build a line of information for each pulse that corresponds to the equivalent lines of a TENEX "MEMSTAT" command.

osrd1% 34k

entry type - procedure address

procedure function (brief) -

This procedure is used to read one word in the address space of the target process.

when called -

This procedure will be called whenever a LM or a DD routine wishes to examine (for whatever reason) a word in the address space of the target process.

arguments -

1st argument: the address of the word the LM or DD wishes to read

results -

1st result:

a value (n) indicating the success or failure of this routine as follows:

n = 0: word read successfully

n < 0: invalid address passed to this routine

n > 0: address passed to this routine represents a non-existent page in the target process

2nd result:

the contents of the addressed word, or 0 on error conditions

error conditions -

Any error conditions detected by this procedure should either be dealt with by this procedure or translated into the appropriate error return.

discussion -

The function of this procedure is to read and return the contents of the passed address in the target process' address space.

csrcmw 341

entry type - procedure address

procedure function (brief) -

This procedure is used to read one or more words in the address space of the target process.

when called -

This procedure will be called whenever a LM or a DD routine wishes to examine (for whatever reason) one or more words in the address space of the target process.

arguments -

1st argument: the address of the first word to read

2nd argument: the number of words the to read

3rd argument: an address at which to store the read words

results -

1st result:

the number of words read and returned to this routine's caller

2nd result:

a value (n) indicating the success or failure of this routine as follows:

n = 0: words read successfully

n < 0: invalid address passed to this routine

n > 0: address passed to this routine represents a non-existent page in the target process

error conditions -

Any error conditions detected by this procedure should either be dealt with by this procedure or translated into the appropriate error return.

discussion -

The function of this procedure is to read and return the requested words in the target process' address space.

osstat 34m

entry type - coroutine address

procedure function (brief) -

to generate (for the user) the operating system state of a process

when called -

In response to the user giving the status command

arguments -

at OPENPORT time

1st argument

the address of the current output mode record to be used for formatting results

2nd argument

FALSE - to indicate give a short status; TRUE - to give a verbose status

at FILSE time

1st argument

the address of a tool record for a process whose status is desired

2nd argument

the address of a string to receive status information

3rd argument

an indicator of how many leading blanks to insert in front of the string passed as the 2nd argument

results -

at OPENPORT time

NONE

at PULSE time

returns TRUE if more status is available about the same process; returns FALSE to indicate that no more status is available about the process represented by the passed tool record

error conditions -

NONE

discussion -

The purpose of this coroutine is to format strings for presentation to the user concerning the operating system state of a given process.

cswr1w 34n

entry type - procedure access

procedure function (brief) -

This procedure is used to write one word in the address space of the target process.

when called -

This procedure will be called whenever a LM or a DD routine wishes to write (for whatever reason) a word in the address space of the target process.

arguments -

1st argument: the address of the word the LM wishes to write

2nd argument: the value to be written in the addressed word

results -

1st result:

TRUE if the word was written successfully; FALSE otherwise.

error conditions -

Any error conditions detected by this procedure should either be dealt with by this procedure or translated into the appropriate error return.

discussion -

The function of this procedure is to write the passed value at the passed address in the target process' address space.

osmrmw 34c

entry type - procedure address

procedure function (brief) -

This procedure is used to write one or more words in the address space of the target process.

when called -

This procedure will be called whenever a LM or a DD routine wishes to write (for whatever reason) one or more words in the address space of the target process.

arguments -

1st argument:

the address, in the target process' address space, of the first word to write

2nd argument: the number of words to write

3rd argument:

an address, in the debugger's address space, from which to get successive words to write in the target process' address space

results -

1st result:

TRUE if the words were written successfully; FALSE otherwise.

2nd result:

the number of words actually written if the first result is FALSE

error conditions -

Any error conditions detected by this procedure should either be dealt with by this procedure or translated into the appropriate error return.

Discussion -

The function of this procedure is write the requested words in the target process' address space.

ossrch 34e

entry type - procedure address

procedure function (brief) -

This procedure will search the target process' address space between 2 passed addresses (inclusively) for cells that contain the passed value (after both have been masked appropriately).

when called -

This procedure will be called by the LM lnfmem coroutine to perform content searches in the address space of the target process.

arguments -

1st argument: a starting address, in the target process' address space

2nd argument: an ending address, in the target process' address space

3rd argument: value to search for

4th argument: mask to apply to search value and words in target process

5th argument: TRUE for a content search; FALSE for a not content search

results -

1st result: TRUE if this procedure found a word that met the passed requirements; FALSE if not.

2nd result: the address of the found word on success; FALSE otherwise.

3rd result: the contents of the found word on success; indeterminate otherwise

error conditions -

Any error conditions detected by this procedure should either be dealt with by this procedure or translated into the appropriate error return.

discussion -

The function of this procedure is to search the inclusive address bounds, in the target process' address space, for the passed value. Both the passed value, and cells in the target process' address space will be masked (logically ANDed) with the the passed mask, before any compares are performed. This procedure can succeed if the resulting compares are equal if this was a content search; and it can also succeed if the resulting compares are not equal and a not content search was specified.

osgpts 34c

entry type - procedure address

procedure function (brief) -

This procedure is used to obtain the state vector of the target process and to write the obtained state vector in the cells allocated for the state vector in the OSM's dispatch table.

when called -

This procedure will normally only be called by the OSM's ostpte routine.

arguments -

NONE

results -

NONE

(This procedure must write the appropriate cells in the OSM's dispatch table with the appropriate state vector.)

discussion -

This function of this procedure is to obtain the state vector of the target process and to write the obtained state vector in the cells allocated for the state vector in the OSM's dispatch table (see discussion below).

ossbfs

34

entry type - procedure address

procedure function (brief) -

This procedure is used by LM or DD routines to modify the state vector of a target process.

when called -

This procedure will be called when a LM or DD routine wishes to modify the state vector of a target process. Neither LM nor DD routines should modify the state vector directly, but they must use this procedure.

arguments -

1st argument

the address of a new state vector to become the current state vector for the current target process

results -

NONE

discussion -

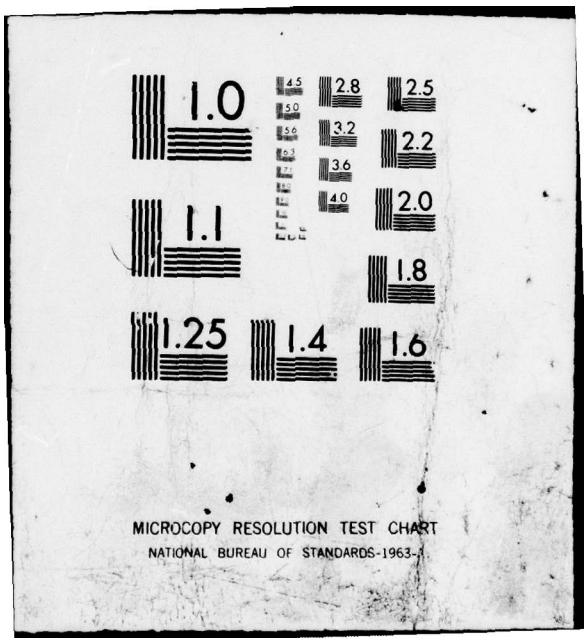
This procedure will modify the state vector of the current target process.

AD-A074 496 SRI INTERNATIONAL MENLO PARK CA F/G 9/2
ON-LINE PROGRAMMER'S MANAGEMENT SYSTEM. ADDENDUM II. PROGRAMMER--ETC(U)
AUG 79 B L PARSLEY, H G LEHTMAN, S KAHN F30602-77-C-0185
UNCLASSIFIED RADC-TR-79-205-ADD-2 NL

3 OF 3
AD
A074496



END
DATE
FILMED
10-79
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

csalos 34s

entry type - procedure address

procedure function (brief) -

NOT IMPLEMENTED YET

when called -

NOT IMPLEMENTED YET

arguments -

NOT IMPLEMENTED YET

results -

NOT IMPLEMENTED YET

error conditions -

NOT IMPLEMENTED YET

discussion -

NOT IMPLEMENTED YET

srels 34t

entry type - procedure address

procedure function (brief) -

NOT IMPLEMENTED YET

when called -

NOT IMPLEMENTED YET

arguments -

NOT IMPLEMENTED YET

results -

NOT IMPLEMENTED YET

error conditions -

NOT IMPLEMENTED YET

discussion -

NOT IMPLEMENTED YET

csrmark 34u

entry type - procedure address

procedure function (brief) -

to remove a breakpoint from the target process' address space

when called -

usually called by the OSM routines that get control when a process becomes the current target process

arguments -

1st argument

the address of a breakpoint record (see DBTDEF) - this record contains sufficient information about a breakpoint to enable this routine to remove the breakpoint from the target process' address space

results -

NONE

error conditions -

NONE

discussion -

This procedure is used to cleanup the access space of a target process so that other routines in the debugger can examine the access space of a target process at will without having to worry about if an address contains code placed there by the debugger (e.g. for breakpoint purposes) or code that is part of the program being debugged. This routine should update the breakpoint record to reflect the current state of inserted breakpoint code in the target process.

dsintrk 34v

entry type - procedure address

procedure function (brief) -

to insert a breakpoint in the target process' address space

when called -

usually called by the OSM routines that get control when a process is about to no longer be the current target process

arguments -

1st argument

the address of a breakpoint record (see DDTDEF) - this record contains sufficient information about a breakpoint to enable this routine to insert the breakpoint in the target process' address space

results -

NONE

error conditions -

NONE

discussion -

This procedure is used to insert any code that an OSM wishes to in the address space of the current target process. E.g., an OSM may want to insert special instructions to indicate breakpoints. When this routine is done, the target process should be in such a state that its execution can be continued later, and if breakpoints are specified, they will take effect. This routine should update the breakpoint record to reflect the current state of inserted breakpoint code in the target process.

oscsacr 34w

entry type - procedure address

procedure function (brief) -

change the current program counter for the current target process

when called -

in response to user commands that specify a new pc for a process

arguments -

1st argument

the new pc value

results -

NONE

error conditions -

NONE

discussion -

This routine should change the program counter of the current target process.

csgpc 34x

entry type - procedure address

procedure function (brief) -

get the pc of the current target process

when called -

as a shorthand mechanism rather than reading the state vector of a process

arguments -

1st argument

the address of a tool record for the process for which the pc is desired

results -

the pc of the concerned process

error conditions -

NONE

discussion -

this procedure is a convenient way of obtaining the pc of a process without having to interpret the state vector, or of obtaining the pc of a process which is not the current target process

dsentl 34y

entry type - procedure address

procedure function (brief) -

to cleanup as the user is done debugging a process

when called -

when the user indicates that [s]he is no longer interested in debugging a process, or if a superior process (in a process tree oriented operating system) kills an inferior process

arguments -

1st argument

the address of the tool record for which we are done debugging the process

results -

NONE

error conditions -

NONE

discussion -

This routine should do any cleanup necessary when the debugger is finished debugging a process.

osstop 34z

entry type - procedure address

procedure function (brief) -

to stop the execution of a process

when called -

any time the debugger is pointed at a process

arguments -

the address of the tool record corresponding to the process whose execution is to be stopped

results -

NONE

error conditions -

NONE

discussion -

This routine should actually stop the execution of the specified process. It should do it in such a way that execution can be resumed subsequently by the osgo routine.

osgo 34aa

entry type - procedure address

procedure function (brief) -

tc resume the execution of a process

when called -

in response to user commands to resume process execution

arguments -

the address of the tool record corresponding to the process whose execution is to be resumed

results -

NONE

error conditions -

NONE

discussion -

This routine should actually resume the execution of the specified process.

OSDCMCSM 34ab

entry type - procedure address

procedure function (brief) -

tc perform any cleanup necessary as the debugger is finished with this OSM

when called -

in response to user commands that indicate that this OSM is no longer necessary

arguments -

NONE

results -

NONE

error conditions -

NONE

discussion -

This OSM will not be used again, and this routine is responsible for any clearing up necessary.

csesta 34ac

entry type - coroutine access

procedure function (brief) -

to obtain information about the last error of the current target process and to format this information for presentation to the user

when called -

in response to certain user commands

arguments -

at OPENPORT time

NONE

at PULSE time

1st argument

the address of a string to get the formatted information

2nd argument

the address of the current output mode record to be used in formatting the above mentioned string

results -

at OPENPORT time

NONE

at PULSE time

FALSE to indicate that no more information is forthcoming

error conditions -

NONE

discussion -

This routine should generate and write to the passed string information pertaining to the last error encountered by the current target process.

csfsta 34ac

entry type - coroutine address

procedure function (brief) -

to obtain information about files known to the processes being debugged and to format this information for presentation to the user

when called -

in response to certain user commands

arguments -

at OPENPORT time

NONE

at PULSE time

1st argument

the address of a string to get the formatted information

2nd argument

the address of the current output node record to be used in formatting the above mentioned string

3rd argument

a file handle on the first file in a range that the user is interested in

4th argument

a file handle on the last file in a range that the user is interested in

results -

at OPENPORT time

NONE

at FILSE time

FALSE to indicate that no more information is forthcoming

error conditions -

NONE

discussion -

This routine implements the equivalent of the TENEX FILSTAT command.

osfsav. 34ae

entry type - procedure address

procedure function (brief) -

to save the address space of a tool on a file

when called -

in response to user commands

arguments -

1st argument

the address of the tool record corresponding to the process whose address space it is desired to save

2nd argument

the address of a string containing a file name on which to save the address space

3rd argument

the address of the current output mode record

4th argument

the address of a string in which to place any messages for the user

results -

NONE

error conditions -

NONE

discussion -

This routine should save the address space of the specified process on the specified file (this is the equivalent of the TENEX SSAVE command).

osich 34af

entry type - simple data structure

data structure meaning -

this is the debugger IDB for the target process that is currently being debugged.

data structure type -

this data structure is a single word in the OSM's dispatch table

discussion -

this data structure will be set and maintained by the DD. It consists of an internal debugger handle for the current target process.

cssvec 34ac

entry type - 50 word data structure

data structure name - PFSTATE

data structure meaning -

this data structure contains the state vector of the current target process.

data structure type -

this data structure is composed of 50 words (under TENEX, the first 16 of these words represent the registers of the target process; the meaning of the rest of the words are currently unspecified.)

APPENDIX - SYMBOLS DEFINED IN THE DEBUGGER LOADER 35

The reader is urged to study the following files for the actual definitions (and meanings) of symbols that are defined in the debugger loader, and are hence available to all debugger modules: 35a

DDTDSP.NLS

LANDSP.NLS

CSIDSP.NLS

DDTDEF.NLS

FEIFACE.NLS

APPENDIX - TARGET PROCESS PRIMITIVES REQUIRED FOR DEBUGGING 36

In order to be able to interactively debug any process, it is necessary that the debugger be able to exercise certain controls over the process. These controls are of the nature of being able to read and write the address space of the process, stopping and starting execution of the process, etc. It is the responsibility of the OSM to provide a standard interface between these functions and the rest of the debugger. 36a

When running under a process oriented operating system, with the debugger at the top of the process tree, the OSM would be able to exercise these functions by operating system primitives that exert control over inferior processes.

When running under a process oriented operating system, with the debugger and the target process both under the control of a common head, some of these functions may be doable by the debugger via operating system primitives directly, and for some the debugger may have to request the common head to perform the function.

If the debugger and the target process are not running on the same machine, or under a common head in the same machine, then it becomes necessary for the OSM to communicate with (procedures in) the target process (or in a process that has control over the target process), to have the needed functions performed.

This appendix will discuss, at a functional level, the procedures needed in a target process to make it debuggable. 36b

function name - START DEBUGGING 36c

arguments:

none

results:

none

function:

This procedure is the first called procedure and it gives the process a chance to do whatever initialization is required to start debugging. (If for some reason this process can not, or is not willing to be debugged, then this procedure gives a FAILURE return.)

function name - READ MEMORY 36c

arguments:

intra-process address

entity to be read (e.g. bits, bytes, words)

number of entities to be read

results:

if successful gives a success return and returns the requested entities; if unsuccessful, gives a failure return and returns an indication as to why it failed and as many of the requested entities as it was possible to return and an indication of how much was returned.

function name - WRITE MEMORY 36e

arguments:

intra-process address

entity to be written (e.g. bits, bytes, words)

number of entities to be written

results:

if successful gives a success return having written the requested entities; if unsuccessful, gives a failure return and returns an indication as to why it failed and having written as many of the requested entities as was possible it returns an indication of how much was written.

function name - FIND CONTENT 36f

arguments:

starting intra-process address

ending intra-process address

entity to be searched for (e.g. bits, bytes, words)

value of entity to be searched for

mask to modify search

results:

if successful, gives a success return and an intra-process address of the found entity; if unsuccessful, gives a failure return and an indication of the reason for failure.

function:

this procedure searches from the starting intra-process address to the ending intra-process address for the first occurrence of the passed value (modifying target memory by the passed mask).

function name = FREEZE (36)

arguments:

process address

results:

none

function:

this procedure freezes the requested entity. freezing an entity stops the execution of code: when an entity is frozen it is not responsive to calls other than calls on debugging functions. (such calls are queued until such time as the entity is thawed.)

function name = THAW (37)

arguments:

process address

results:

none

function:

this is the inverse of freezing.

function name - READ STATE 36i

arguments:

process address

results:

returns a state vector describing the state of the selected process. the content and syntax of the state vector is dependent on which operating system is concerned. the state vector will contain such things as: the registers of the concerned process, where it was executing when it was interrupted, the state of its pseudo-interrupt system (on TENEX), whether or not the process was in single step mode (on machines that support this), etc.

function:

The intent of this procedure is to be able to save the state of a process, modify the state, perform some arbitrary computation, and then restore its original state and resume its prior execution as if none of this had happened.

function name - WRITE STATE 36j

arguments:

process address

state vector

results:

if the state vector of the selected entity is successfully written, then this procedure gives a success return; if the state vector cannot be written, then a failure return is given as well as an indication as to why the state vector could not be written.

function name - SET BREAKPOINT 36k

arguments:

intra-process address

proceed count

procedure address

tflag

fflag

results:

returns the contents of the cell(s) that were replaced in order to set the breakpoint and an indication of how many, and what type of (bytes, words, etc.), cells were replaced.

function:

this procedure sets a breakpoint at the specified address. when a breakpoint is encountered it either takes or fails. the passed flags (tflag for the breakpoint takes, and fflag for the breakpoint failing), indicate whether or not the debug process is to be notified (discussed below) if the breakpoint takes or fails.

the proceed count indicates how many times the instruction at the passed intra-process address should be executed before the breakpoint takes; the passed procedure address is a procedure to be called in the target process to decide whether or not to take the breakpoint. the precedence of the proceed count and the called procedure will be detailed later.

function name - REMOVE BREAKPOINT 36l

arguments:

intra-process address

results:

none

function:

this procedure removes the breakpoint at the specified intra-process address.

function name - GET SYMBOL TABLE ADDRESS 36m

arguments:

intra-process address

results:

returns the intra-process address of the symbol table for the passed entity and the length of the symbol table and the type of the symbol table

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.