load file: system file error

i tried several times to load file a nls file from another directory but each time i received the message "system file error", i then tried to get the file by jumping to the file by giving the jump address command the file name in link form, this worked fine, --jon, 1 load file: system file error

(J24238) 17-OCT-74 14:23;;;; Title: Author(s): Jonathan B. Postel/JBP; Distribution: /BUGS([ACTION]) ; Sub-Collections: SRI-ARC BUGS; Clerk: JBP;

anthropomor... 8,34

brevity is indeed important, but not as important as clarity. anthropomorphasizing (?) can sometimes aid the causes of both clarity and brevity, as well as making the material more enjoyable to read -- an oft times overlooked aspect of our documentation. 1

(J24239) 17-OCT-74 14:46;;;; Title: Author(s): Kenneth E. (Ken) Victor/KEV; Distribution: /SRI-ARC([INFO-ONLY]) ; Sub-Collections: SRI-ARC; Clerk: KEV;

Pete Tasker to visit 18 Oct 74 at 1330

Jim, Dick: Pete Tasker will visit ARC at 1:30 tomorrow, Friday. He'd like to see AKW in action. He is a MITRE guy (friend of Jean Iseli's), whose current work is associated with (like) COTCO out on Oahu, in a "loose framework" there. I gather that he is doing a study in an environment similar to what COTCO was to be aimed at, and that he has evolved toward wanting to consider/experiment with services more towards a fuller AKW than just message system. 1

This visit would (to me) be classed as "potential Utlity client". He may bring a second MITRE guy along. I told him that JCN and I would probably meet initially just to hear his story; probably have a third ARC (application) guy to hear, too, who could then give him demo and discussion -- closing by base touching with DCE/JCN before he leaves. 2

Dick: We can easily include you (or Development person) if you want. 3

Regards, Doug 4

Pete Tasker to visit 18 Oct 74 at 1330

(J24240) 17-OCT-74 16:50;;;; Title: Author(s): Douglas C. Engelbart/DCE; Distribution: /JCN([ACTION]) RWW([ACTION]) SRI-ARC([INFO-ONLY]) ; Sub-Collections: SRI-ARC; Clerk: DCE;

HGL 17-OCT-74 17:53 24241

A test of Journal delivery

Does it work? 1

A test of Journal delivery

(J24241) 17-OCT-74 17:53;;;; Title: Author(s): Harvey G. Lehtman/HGL; Distribution: /BUGS([ACTION]) JDH([INFO-ONLY]) HGL([INFO-ONLY]) ; Sub-Collections: SRI-ARC BUGS; Clerk: HGL;

Missing Indeces: All the Links in the Attached Group Yield the Message File Not Online

From JUL 72 Thru JUL 73
(:ebtz) 1

AUTHOR INDEX
(catalog, arcjaincnl, 0:wD) 1a

NUMBER INDEX
(catalog, arcjnincnl, 0:wD) 1b

A-F TITLEWORD INDEX
(catalog, arcjtafincnl, 0:wD) 1c

G-O TITLEWORD INDEX
(catalog, arcjtgoincnl, 0:wD) 1d

P-Z TITLEWORD INDEX
(catalog, arcjtpzincnl, 0:wD) 1e

From JUL 73 Thru DEC 73
(:ebtz) 2

AUTHOR INDEX
(catalog, barcjaincnl, 0:wD) 2a

NUMBER INDEX
(catalog, barcjnincnl, 0:wD) 2b

A-F TITLEWORD INDEX
(catalog, barcjtafincnl, 0:wD) 2c

G-O TITLEWORD INDEX
(catalog, barcjtgoincnl, 0:wD) 2d

P-Z TITLEWORD INDEX
(catalog, barcjtpzincnl, 0:wD) 2e

From JAN 74 Thru JUN 74
(:ebtz) 3

AUTHOR INDEX
(catalog, carcjaincnl, 0:wD) 3a

NUMBER INDEX
(catalog, carcjnincnl, 0:wD) 3b

A-F TITLEWORD INDEX
(catalog, carcjtafincnl, 0:wD) 3c

G-O TITLEWORD INDEX
(catalog, carcjtgoincnl, 0:wD) 3d

P-Z TITLEWORD INDEX
(catalog, carcjtpzincnl, 0:wD) 3e

From JUL 74 Thru DEC 74
(:ebtz) 4

AUTHOR INDEX
(catalog, darcjaincnl, 0:wD) 4a

NUMBER INDEX
(catalog, darcjnincnl, 0:wD) 4b

A-F TITLEWORD INDEX
(catalog, darcjtafincnl, 0:wD) 4c

G-O TITLEWORD INDEX
(catalog, darcjtgoincnl, 0:wD) 4d

P-Z TITLEWORD INDEX
(catalog, darcjtpzincnl, 0:wD) 4e

Missing Indeces: All the Links in the Attached Group Yield the Message File Not Online

(J24242) 17-OCT-74 21:26;;;; Title: Author(s): Dirk H. Van Nouhuys/DVN; Distribution: /JCN([ACTION]) JCP([ACTION]) KIRK([ACTION]) ; Sub-Collections: SRI-ARC; Clerk: DVN;

Trip of 7-9 oct 74 to bbn & compass

< POSTEL, TRIP.NLS;2, >, 10-OCT-74 15:15 JBP ;;;; 1

Jim White and i have just returned from meetings with (1) Vint Cerf, on internet protocol; (2) Bob Thomas and Rick Schantz, on RSEXEC and procedure call protocol; and (3) Steve Warshall and Bob Millstein, on Works Manager and procedure call protocol. 1a

It is my opinion that the procedure call protocol is the correct approach to the process to process level of interactions between the nsw components. 1b

There is general agreement on the form and function of the procedure call protocol, although there needs to be some work on the environment control package. It is expected that a updated set of documents will be ready at the end of October whic describe the proceture call protocol and the support and environment control packages. These new documents will be close enough to the final form that others can use them for preparing proceedure packages. 1b1

An initial implementation of the procedure call mechanism was started by Jim for the NLS split into front and back ends. The discussions have lead to a decision that the implementation should be made in a language that will easily run on any tenex, so the implementation may be changed from L10 to BCPL. 1b2

It is my current view that the Internet Protocol will not be sufficiently implemented and tested (debugged) of use in the first year nsw. 1c

I do believe that the Internet Protocol is likely to offer significat advantages in performance over the existing host to host protocol, therefore we will attempt to implement the procedure call protocol in such a way that it can be easily switched from one to the other underlying protocol. 1c1

I expect that the internet protocol development continue and that experiments be carried out to show the thruput and delay characteristics of each of the protocols (current host-host, and internet). These studies should be completed by the end of the first year of nsw, that is 30 June 1975. 1c2

An area of concern is the ADR tasks, rumors have come my way that ADR is not comming up to speed as fast as desirable on ARPANET technology and protocol considerations. There is a lot of help available in the boston area, and if necessary i would be willing to spend some time speeding their education. It is crucial for nsw that the B4700 interface be ready as soon as possible. 1d

Bob Thomas expresses his interest in the nsw development and his willingness to comment on any plans for protocols or software. I urge all nsw participants to receive the advantage of Bob's exprience with RSEXEC and TENEX. 1e

Trip of 7-9 oct 74 to bbn & compass

(J24243) 18-OCT-74 10:31;;;; Title: Author(s): Jonathan B. Postel/JBP; Distribution: /JBP([ACTION]) ; Sub-Collections: SRI-ARC; Clerk: JBP;

File name write up in Help

The word field is confusing,since altmode doesn't work in TNLS it too is confusing, 1

File name write up in Help

(J24244) 18-OCT-74 10:36;;;; Title: Author(s): JOAN HAMILTON/JOAN;
Distribution: /FDBK([ACTION]) KIRK([ACTION]) ; Sub-Collections:
SRI-ARC; Clerk: JOAN;

< POSTEL, NSW-PROT-IMPL-PLAN.NLS;4, >, 18-OCT-74 13:06 JBP ;;;; 1

The NSW project requires protocols and service routines to be implemented at several levels. 1a

There needs to be a more effective host-to-host protocol. 1b

The INTERNET protocol is a good candidate, but there must be a test implementation and a comprehensive comparative measurement against the standard host-to-host protocol. 1b1

Beyond this the NSW will use a Procedure Call Protocol (PCP) for communication of service requests between the NSW modules.This will require implementation of the PCP mechanism in each of the computers that participate in the NSW. 1c

There will also be a set of standard service packages that may communicate (via PCP) with other modules in NSW. This set of service packages includes 1d

a File Manipulation Package 1d1

This package contains functions to move files between work-spaces either on the same system or between systems. This package implements most of the "blackboxes" suggested by compass. 1d1a

a Remote Job Entry Package 1d2

This package contains functions to submit and retrieve files from the batch processing facility of this system. 1d2a

an Executive Package 1d3

This package contains functions to report status information about the system or tasks operating ont the system. For example accounting information. 1d3a

and for the satellite a Front End Control Package 1d4

This package contains functions for the control of the users terminal from the works manager or tools. The type of thing envisioned is indicating how a screen should be divided into windows. 1d4a

ARC has the responsibility for specifying the protocols and service packages for NSW. In the implementation of these packages however ARC needs assistance. It is our understanding that BBN has been funded by ARPA to assist in the development of NSW protocols in cooperation with ARC, We realize that BBN has some specific direction from ARPA on which protocols to expend effort on, we are asking for assistance as available in the areas listed below, 1e

The following indicate our current perception of the protocol implementations needed for the NSW initial phase, 1f

Here we are discussing the TENEX implementation of the various packages, we expect other parties to implement these protocols for other systems, 1g

Host-to-host protocol 1g1

The Internet Protocol as specified by Cerf must be implemented and tested in comparison with the standard protocol, We expect that BBN an SU (and perhaps others) will carry out these activities, 1g1a

A comprehensive test program may require the implementation of Telnet and File Transfer protocol interfaces to Internet protocol, 1g1a1

Perhaps constrain the implementations of the standard protocol to conform to certain buffering and allocation policies, We expect that these policies will be specified by ARC and installed by BBN, 1g1b

Procedure Call Protocol 1g2

Implementation of Procedure Call Protocol by ARC with advice from BBN, 1g2a

File Manipulation Package 1g3

Specification by ARC and implementation by BBN, 1g3a

Remote Job Entry Package 1g4

Specification by ARC and implementation by BBN, 1g4a

Executive Package 1g5

Specification by ARC and implementation by BBN, 1g5a

Front End Control Package 1g6

Specification and implementation by ARC, 1g6a

Please note that in all cases the specification process will involve review for comments and suggestions by interested parties including BBN. 1h

Protocol implementation plan draft

(J24245) 18-OCT-74 13:08;;;; Title: Author(s): Jonathan B. Postel/JBP; Distribution: /RWW([ACTION]) JEW([ACTION]) ; Sub-Collections: SRI-ARC; Clerk: JBP;

Express Log suggestion

When I Express log in, I would like to know from the beginning at what time I wll be logged out. 1

Express Log suggestion

(J24246) 18-OCT-74 13:13;;;; Title: Author(s): N. Dean Meyer/NDM;
Distribution: /FDBK([ACTION]) ; Sub-Collections: SRI-ARC; Clerk:
NDM;

MINUTES OF DOCUMENTATION MEETING OF 10-14-74: Status of Documentation, Plans for Introductory Hardcopy for Help, Plans for Something for Learners to Read.

ATTENDEES: RWW, DVN, POOH, KIRK 1

Status of various documents stands as follows: 2

Cue-card. Finished. A few small errors, available from Documentation shelf, Jim Bair took a supply with him when he left of the east Sunday to present NLS-8 to various user groups. 2a

TNLS-8 Primer: Bair took a draft (mjournal,23911,) acceptable to him. A COM version is being processed at DDSI. JCN may still have input. 2b

Command Summary. A version reflecting the language as of October 6th is online (Userguides, Commands) and (mjournal,23912,) and Jim Bair took a supply with him. We will produce a new version when the languages is realy frozen and consider COMing it. 2c

NLS-8 Equivalents of NLS-1 Commands. Finished as (mjournal,23913,). Jim Bair took a supply with him. 2d

Line Processor User's Guide. Jim Bair took a draft with him that did not include all recent suggestions from MEH, RWW,DIA 2e

The Glossary. we considered Dick's suggestions Ann, is now going over the Glossary with an eye toward making it more readalbe to new users. 2f

we agreed that procedures that differ between ARC and Office-1 will be written up for Office-1. (e.g. login, feedback, guest accounts) 2f1

We agreed that the plan to pull the branch (documentation, help, how) out of the Help Data Base with minor modification to serve as an introduction to the Glossary did not work. 2f2

At about this point in the meeting Dick left. Before he left the others askd how much worktime we had to devote to the efforts discussed here. He suggested about person-month. 2g

Those of us remaining strove to define what minimum knowledge a user needs go ahead and learn from Help. We agreed that the minimum included live experience, and we would assume that anyone trying to learn from Help had access to the Primer. We agreed that Ann should try to assemble a two-page document giving the other information necessary. It will include a anotated diagram of the syntax of an NLs Command which Dirk will contribute, the figure of NLS structures for Help, and a separate one-page write up on TNLS addressing which Dirk will contribute. 2h

These writeups will assume that the user begins at the point where she sees the TENEX harald. We considerd a wrteup that listed all the ways you can reach NLS, but in a later conversation Dick discouraged that idea 2i

Drafts were due Friday10/18 but didnot makeit. 2j

The question remains of providing in a reasonable time a document for people who want to sit down and read and get a general notion of NLS useful in their learning process. Dirk will endeavor to create such a document based on Help. As a first cut he assembled the following rough list of topics from (documentaton,help,how) and welcomes suggestions for addtions and omissions: 2k

How to use NLS:
You use NLS by typing in commands. Commands begin with verbs such as "Insert" or "Substitute", or "Delete". They write in, locate, transform, or disseminate text from the computer. To use NLS, you must understand commanding. See also: NLS. 2k1

Getting Help:
1) strike ? at any point in an NLS command for a list of alternatives currently available to you.
2) hold down the <CTRL> button and hit q, at any point, for an explanation of your current alternatives.
Method 2 puts you into the Help command repeat mode until you hit CD (Command Delete <CTRL-X>) See also: HELP, CD, REPEAT. 2k2

questionmark
##<questionmark>## 2k3

<CTRL-Q>
##<CTRL-q>## 2k4

Getting just the syntax of a command <CTRL-S>
If you hold down the CTRL key and type s, you will get the command syntax for the command which you are currently using. 2k5

When help fails
Novices should feel free to connect to experienced users and ask questions. Keeney, Kelley, van Nouhuys, Beck, and Bair are particularly open to connecting. Also, sendmail to ident FDBK explaining what went wrong. 2k6

Subsystems: entering and leaving
When you enter NLS, you begin in the Base subsystem, A number of other SUBSYSTEMs are available, To leave NLS or any other SUBSYSTEM, use the Quit command, To goto another NLS SUBSYSTEM, use the Goto SUBSYSTEM command, See also: SUBSYSTEM 2k7

A list of subsystems with their uses, 2k8

Commanding:
##<command>## 2k9

Nominal-Verbal rhythm, Options, etc, 2k9a

Pointing to information: addressing and bugging
In TNLS, pointing moves an invisible Control Marker (CM) to a specific character in a statement within a file, You point in this way whenever a command asks for an ADDRESS (prompts you with A:), In DNLS, you can also point by bugging with the mouse, If a link appears in the text of a file, you may point at the link and then indicate to the system that you want the command to act at the place named in the link, 2k10

Reading and viewing information:
You can read all NLS files whose name you know, except files whose access has been specifically restricted, You call files with the Load File Command, After you have loaded it, you can move around within its structure by pointing, view it in different ways with viewspecs, and print or output it for reading, See also: pointing, information, For DNLS, See also: viewing 2k11

accessing files:
Wherever an ADDRESS (A:) is prompted, you can go to a particular file whose FILEADDRESS you know--type it in, You can also use the Load File command to open a file for read or write access, You can insert into a Statement a LINK that points to a file which can then later be used to access the file by pointing to the link, A record of the files you have been in during your current NLS session, the file-return stack, provides another method of accessing those files easily, When you use the Create File command in NLS, the new file is immediately loaded for you, Access to files may be protected, See also: prompts, creating, modifying, 2k11a

moving around in files and printing on your terminal in TNLS:

The family of Jump and Print commands are used to view information in TNLS. Jump to Address is the basic TNLS pointing command. Other Jump commands point to a character within a statement; some point to files; and some point to statements by their structural position. See also: pointing, file, structural, Jump Address TNLS
##<printing>## 2k11b

Hardcopy printing and formatting
##<hardcopy>## 2k11c

% 2k11d

Writing, creating and modifying information:
you can create new files, copy all or selected parts of existing files, insert text by typing into existing files, and edit existing text. Access for these operations may be restricted. See also: commanding, pointing, viewing, information, file. 2k12

The Insert command allows you to create information.
##<insert>## 2k13

creating files:
##<create>## 2k14

handling whole files:
NLS provides many commands that deal with whole files allowing you to incorporate modifications, delete modifications, send them to people, ##<%archive them on tape,>## delete them, and transfer them from one directory or site to another, and return to recent files you have accessed. See also: accessing, creating, modifying, sending, updating, directory, site. 2k15

modification file:
you can edit files temporarily or permanently using the NLS Base subsystem. The name you were logged in under when you made the modifications precedes the filename in parentheses. Its version number is the same as the NLS file, but its extension is .PC; (for Partial Copy) instead of .NLS;. This file disappears when the Update, or Delete Modification commands are used. The Update command incorporates the changes permanently into the NLS file. To delete the modifications you have made since the last Update, use the Delete Modification command. 2k16

correcting errors:

To escape from a command you have started, type <CTRL-x>. Inside a TYPEIN (following T:), to backspace and delete one character, type <CTRL-a>; to backspace and delete back to the previous space, type <CTRL-w>. The commands people use most often to correct errors in text that is already online are Substitute and Replace See also: Substitute, Replace, BW, BC, CD, CTRL-x, CTRL-a, CTRL-w 2k17

% 2k18

Sending mail:
##<sendmail>## 2k19

Hardcopy printing and formatting:
You may print your NLS files at your terminal, at a line printer at ARC, at a printer at your site if it is available, or through COM (Computer output to Microfilm. COM offers offset with graphic arts quality type. A set of embedded directives allows you to design formats flexibly. See also: sendmail offline. 2k20

Profile defining: the useroptions subsystem
##<useroptions>## 2k21

Programming for users:
##<programs>## 2k22

Addressing 2k23

Bug 2k23a

SOURCE<DESTINATION CONTENT 2k23b

Address elements 2k23c

Information Hierarchy (bit to site, from Help) 2k24

Printing 2k25

At your terminal 2k25a

Quickprint 2k25b

Output Printer 2k25c

Output COM 2k25d

MINUTES OF DOCUMENTATION MEETING OF 10-14-74: Status of Documentation, Plans for Introductory Hardcopy for Help, Plans for Something for Learners to Read.

NLS files vs Sequentail Files 2k26

Filter Options and Conten Analyser 2k27

BsYs 2k28

MINUTES OF DOCUMENTATION MEETING OF 10-14-74: Status of Documentation, Plans for Introductory Hardcopy for Help, Plans for Something for Learners to Read.

(J24247) 18-OCT-74 13:49;;;; Title: Author(s): Dirk H. Van Nouhuys/DVN; Distribution: /JOAN([ACTION] Please add this to the dirt notebook) DIRT([INFO-ONLY]) ; Sub-Collections: SRI-ARC DIRT; Clerk: DVN;

This is a test of journal delivery

Another test of the journal delivery. 1

This is a test of journal delivery

(J24248) 18-OCT-74 14:11;1, >, 18-OCT-74 14:21 XXX ;;;; Title:
Author(s): Harvey G. Lehtman/HGL; Distribution: /BUGS([ACTION]) JDH(
[INFO-ONLY]) ; Sub-Collections: SRI-ARC BUGS; Clerk: HGL;

LP Problems

I'm on the Delta-Data--Line-Processor via the high-speed line and TIP to ARC running the running version of NLS. I have a horizontally split screen with viewspec o in the bottom window, no statement numbers, blank lines on.

I've been getting the error message "Illegal number of blanks requested in CLINE". Also, when I do an edit which shortens a statement, it doesn't erase the line which should then be blank (e.g. the line above the one which was blank before the edit). 1

LP Problems

(J24249) 18-OCT-74 15:27;;;; Title: Author(s): N, Dean Meyer/NDM; Distribution: /FDBK([ACTION]) CHI([INFO-ONLY]) DIA([INFO-ONLY]) ; Sub-Collections: SRI-ARC; Clerk: NDM;

Notes on OFFICE-1 Swapping and Response

This is a collection of notes by DIA concerning the response and swapping problems at OFFICE-1 as of 10/18/74. 1

The problem, as I currently see it is: 2

My statistics were taken when OFFICE-1 had 192K, and was badly overloaded. (10/7/74) But I think my comments hold even with 256K. 2a

The system is much better than with 128K in terms of efficiency and CPU utilization, but still not as good as the ARC system. Notice: 2b

Parameter FL (frustration level) is high (10-15). Indicates that users are waiting a great deal for the system to perform for them. Users don't need to be told that! FL should be about 5. 2b1

%SYS is large (about 90%) and indicates that a very small amount of time (10%) is spent actually executing user program code. %SYS is about 70-80% on ARC (2-3 times better). 2b2

I/O wait IOW is high (20%), but worst of all, most of the IOW time is spent with the drum free -- i.e. system is waiting on the disk (%DW about 20% also). This indicates that the drum is not doing its job. 2b3

There are other indications that the system is overloaded for its configuration, and that swapping in the chief bottleneck. 2b4

It is pretty clear to me that 1) system efficiency and 2) user response would be improved by providing a better swapping mechanism. 2c

The drum is not doing its job because it is not large enough. I think it now holds only 600 pages. I would guess it should be more like 2000 pages. 2c1

As a result programs are spending lots of time waiting for disk pages to come in. 2c2

Drum transfers take about 10 ms. per page on the OFFICE-1 drum. I don't know what a disk transfer takes, but it must be in the range 50-80 ms. A program must wait for transfers that are ahead of it in the queue, hence (approximately) multiply these times by the queue length for the time a program must wait. These figures affect system efficiency by enlarging IOW -- however lots of balance set jobs would be a hedge against the chances of no runnable jobs. But these figures affect user response directly since the user's program must wait that long for each page fault -- and there are many page faults during any typical NLS activation. HENCE, for a responsive system, fast swapping is crutial. 2c3

Options for better swapping facility are: 3

The first option is to obtain a better swapping device to replace the DEC RM10B (current drum). 3a

The problem is, what? It is difficult to obtain a Bryant drum like ARC's, and that would be questionable as far as maintenance etc. goes. A more likely possibility would be the swapping - fixed had disk device that ISI is using. Same questions about how Tymeshare would like that, tho. 3a1

Looks to me like the others alternatives should be pushed. 3a2

Second, forget the drum altogether and try just using disk packs. 3b

(andrews,packs) from April 1972 contains calculations on service that could be expected from a disk pack system, assuming good algorithms and alocation. 3b1

That file indicates that with three disk controllers and two drives on each controller, a disk queue length of 2 on each controller (total queue length = 6), that the total time to read a page (including queue wait) is about 65-70 ms. 3b2

I won't go into additional doodling I have done to come to my conclusions. 3b3

But my conclusions are that it may be close choosing between this option and the next (third option). However, I think this option may hurt user response more than the next option, simply because the time a program waits for a page will probably be longer. More expensive too? 3b4

In any event, I don't think it would be good to try it with only two disk controllers. Response may even be worse than it is now! To do this right, I would even suggest three controllers and three packs on each controller (i.e. three moving arms on each controller). Three arms is more efficient than two. It doesn't pay to go above three however, except to get more disk space. 3b5

Third, expand the current drum. 3c

The current RM10B does not have the swapping characteristics that devices such as ARC's Bryant drum have. Namely, their effeciency is very low and constant. The system can count on an average of about one transfer per rev max. It takes about 10 ms. to get a page from it, no matter the queue length or what. 3c1

A swapping device like the Bryant drum has more pages pass under the heads per rev and can 'skip' from one track to another between sectors. The system can count on many transfers per rev, by ordering the hardware-readable queue to correspond with the order on the drum. The result is that the device has low efficiency at low queues, but the efficiency goes up as the queue length goes up! Our empirical experience with the Bryant drum is that it takes 30 ms. total wait time to get a page from it no matter what the queue length! (Faster at low queue length, e.g. 18 ms when q=1). 3c2

Note that the DEC RM10B is actualy faster - 10 ms vs. 18 ms. with a queue length of 1. But when the queue length gets longer, the Bryant wins big. Say, with a queue of 5, DEC takes 50 ms and Bryant takes 30 ms. The moral is that if your drum queue length is greater than 3, you should have a Bryant-type device. 3c3

My observation is that ARC's drum queue length is not often very much over 3-4 so the DEC drum is a reasonable device if it had the capacity needed and the system load were controlled so that the drum queue did not get out of hand. 3c4

The drum capacity can be increased - up to 4 'drums' per drum controller. I would strongly suggest that OFFICE-1 folks do that as soon as possible. It would not involve even a software change (except for drum bit tables etc.) If done soon enough, perhaps there will be time to re-evaluate before the configuration of OFFICE-2 etc is/are firm. 3c5

Notes about maximizing disk utilization, swapping on them or not: 4

I have my doubts about the performance of the OFFICE-1 disk system, as it stands now. I do not have all the statistics because of a problem in the disk driver code, so I can't say for sure... 4a

Here is a summary of things the disk system should have/do in order to be efficient: 4b

File pages should be spread evenly over all packs. TENEX originally tried to assign related pages to the same pack but that is nonsense. We have modified the system to assign new pages randomly over packs. We also attempt to assign new pages to the center tracks -- thinking that most transfers are to 'young' pages, and that most transfers should ideally be to the center tracks to minimize arm movement. We even implimented but never tried (I think) a system of loading the entire disk from tape , allocating edge tracks first - keeping 'old' pages out of the center. Our packs are so nearly full tho, that this would not make much difference. 4b1

You want three if possible, but certainly at least two drives per controller, and you want software that will position the head(s) on one/two drives while doing a transfer on the other. 4b2

You want a good algorithm for selecting which of the transfers in the queue for a given pack will be transferred next. ARC's system currently takes the transfer closest to the current head position (minimum head movement). 4b3

You want the disk software to do all reads before writes. No programs wait directly for writes to occur. All reads have a program waiting for them. 4b4

One additional note: 4c

The users of ARC's system benefit from the fact that there are two disk controllers -- disk transfers take place faster. To get the most out of a PDP-10 for NLS usage, I would suggest 256K, swapping device, and two disk pack controllers with three drives on each. 4c1

Notes on OFFICE-1 Swapping and Response

(J24250) 18-OCT-74 15:54;;;; Title: Author(s): Don I, Andrews/DIA; Distribution: /RLL([ACTION]) ; Sub-Collections: SRI-ARC; Clerk: DIA; Origin: < ANDREWS, OFFICE-1-SWAP.NLS;3, >, 18-OCT-74 15:50 DIA ;;;;####;

FORGETFULLNESS

TODAY, IF YOU HAPPEN TO SEE IT, COULDD YOU PLEASE GIVE IT TO SANDY FOR PICK UP BY ME WHEN I COME IN, THX...[GEOFF]

FORGETFULLNESS

(J24252) 20-OCT-74 20:41; Title: Author(s): Geoffrey S. Goodfellow/GSG; Distribution: /SRI-ARC; Sub-Collections: NIC SRI-ARC; Clerk: GSG;

The Best One Can

re 24252,: ive located today....but cant seem to get a hold on it, wh and/or if i do ill certainly deliver it to sandy for you,, keep hoping Jim 1

The Best One Can

(J24253) 21-OCT-74 09:29;1, >, 21-OCT-74 10:43 XXX ;;;; Title: Author(s): James C. Norton/JCN; Distribution: /SLJ([INFO-ONLY]) KEV([INFO-ONLY]) ; Sub-Collections: SRI-ARC; Clerk: JCN;

Confused

Geoff, What Is the 'it' to which you are referring? Is it Host Addr 180? If so, this is a new TIP at ISI but as yet they have not responded to my request for a name, so until I get one it is just 180, 52 and 244 are also on the same IMP, (all decimal - sorry!), Jake 1

Confused

(J24254) 21-OCT-74 10:30;1, >, 21-OCT-74 10:45 XXX ;;;; Title: Author(s): Elizabeth J. (Jake) Feinler/JAKE; Distribution: /GSG([ACTION]) ; Sub-Collections: SRI-ARC; Clerk: JAKE;

Requested change in screen update procedure

NLS currently repaints both frozen statements and the dotted line on every update, whether necessary or not. Would be nice (esp. for 1200 baud LP's) if they were left alone. Also, the last statement fragments are repainted when not necessary -- pathetic when the last statement covers half the screen! 1

Requested change in screen update procedure

(J24255) 21-OCT-74 11:21;1, >, 21-OCT-74 11:38 XXX ;;;; Title: Author(s): Don I. Andrews/DIA; Distribution: /FDBK([ACTION]) CHI([INFO-ONLY]) KJM([INFO-ONLY]) ; Sub-Collections: SRI-ARC; Clerk: DIA;

Form System Design Sent to Bill Carlson

Augmentation Research Center
Stanford Research Institute
Menlo Park, California 94025

William E. Carlson
USAF AFDSC/SFP
The Pentagon
Washington, D.C. 20330

Dear Bill:

Dick Watson told me to send you information about the Forms system design created by Elizabeth and me about a year ago. I have therefore included copies of ARC Journal documents (21808,) on the Form system itself and (22394,) on the use of the Datacomputer in the system. While most of the design remains viable today, it should be read with the following understandings: 1

1. Because of various shifts in priorities and shortages in programming resources, the design was never implemented fully, though various tests were carried out. The designs, however, remain essentially valid. 1a

2. The system was designed with the proposed MST enviroment in mind. Hence the emphasis on the use of the Datacomputer for the Data Management part of the system. Note that the use of the Datacomputer is not essential and that the discussion of the advantages and disadvantages of the Datacomputer was made almost a year ago. 1b

3. Implementaton of the new NLS file system with extended property list structure had not been expected to occur before the implementation of the form system. As noted, such a file system would make it easier to construct the form system as designed. The property list structure is essential to our new line drawing graphics system and will be implemented soon. Thus the form system should take advantage of its features. 1c

If you have any questions of the enclosed design documents, Elizabeth or I will try to answer them. 2

Sincerely,

Harvey G. Lehtman
Augmentation Research Center

Form System Design Sent to Bill Carlson

(J24256) 21-OCT-74 13:28;;;; Title: Author(s): Harvey G. Lehtman/HGL; Distribution: /WEC([ACTION]) RWW([INFO-ONLY]) EKM([INFO-ONLY]) DVN([INFO-ONLY]) ; Sub-Collections: SRI-ARC; Clerk: HGL; Origin: < LEHTMAN, FORMS.NLS;1, >, 21-OCT-74 11:54 HGL ;;;;####;

JDH 21-OCT-74 13:02 24257

New Version of PREVIEW at Office-1 Tomorrow Evening

I expect to bring up a new version of PREVIEW at Office-1 on Tuesday evening (10-22). 1

The Main changes are: 2

y viewspec works 2a

Set NLS protection works 2b

The substitute command has been changed: (don't have a fit 'til after you read this) 2c

The "Finished?" question allows a "Show status" in addition to "Yes" or "No". 2c1

The only drawback is a seemingly unavoidable prompting glitch in TNLS which leaves your printout looking like "(Finished?) S/Y/N: Y/N:" when you take the "yes" or "no" options. 2c2

Now you can have a fit. 2c3

Other bugs/mods are listed in (MJOURNAL,24217,1:w). 3

New Version of PREVIEW at Office-1 Tomorrow Evening

(J24257) 21-OCT-74 13:02;;;; Title: Author(s): J. D. Hopper/JDH;
Distribution: /JHB([ACTION]) RLL([ACTION]) KWAC([ACTION]) ;
Sub-Collections: SRI-ARC KWAC; Clerk: JDH;

&SRI-ARC 31-OCT-74 14:54 24258
ARC Rev. 1 NOV 74

The L10 Users' Guide has been updated. New offline copies are available on the shelves in room J2028 or sendmsg to Weinberg at SRI-ARC and request an offline copy. To read the most recent version of this document, jump to link:
<USERGUIDES,L10-Guide,1:w>

TABLE OF CONTENTS

INTRODUCTION..2

PART ONE: Content Analyzer patterns.................................3

Section 1: Introduction...3A
Section 2: Patterns...3B
Section 3: Examples of Content Analyzer Patterns................3C
Section 4: Using the Content Analyzer...........................3D

PART TWO: Introduction to L10 Programming..........................4

Section 1: Content Analyzer Programs............................4A
Introduction..4A1
Program Structure...4A2
Procedure Structure...4A3
Example:..4A4
Declaration statements..4A5
Body of the Procedure...4A6
Programming Style: File Structure...............................4A7
Using Content Analyzer Programs.................................4A8
Problems..4A9
Section 2: Content Analyzer programs: Modifying.................4B
Introduction..4B1
String Construction...4B2
Example:..4B3
More Than One Change Per Statement..............................4B4
Controlling Which Statements are Modified.......................4B5
Problems..4B6

PART THREE: Basic L10 Programming...................................5

Section 1: The User Program Environment.........................5A
Introduction..5A1
Sequence Generator..5A2
Formatter...5A3
Content Analyzers...5A4
User-Written Sequence Generators................................5A5
Section 2: Program Structure....................................5B
An NLS user program consists of the following...................5B1
An example of a simple L10 program is provided here.............5B2
Section 3: Declarations...5C
Introduction..5C1
Variables...5C2
Simple Variables..5C3
Arrays..5C4
Text Pointers...5C5

Strings..5C6
Referenced Variables......................................5C7
Declaring Many Variables in One Statement...............5C8
Declaring Locals..5C9
Section 4: Statements..5D
Introduction..5D1
Assignment..5D2
IF Statement..5D3
CASE Statement..5D4
LOOP Statement..5D5
WHILE...DO Statement......................................5D6
UNTIL...DO Statement......................................5D7
DO...UNTIL/DO...WHILE Statement..........................5D8
FOR...DO Statement..5D9
BEGIN...END Statement....................................5D10
EXIT Statement...5D11
REPEAT Statement...5D12
DIVIDE Statement...5D13
PROCEDURE CALL Statement.................................5D14
RETURN Statement...5D15
GOTO Statement...5D16
NULL Statement...5D17
Section 5: Expressions.......................................5E
Introduction..5E1
Primitives..5E2
Operators...5E3
Expressions...5E4
Section 6: String Test and Manipulation.....................5F
Introduction..5F1
Current Character position (CCPOS)........................5F2
FIND Statement..5F3
FIND Patterns...5F4
String Construction.......................................5F5
Example:..5F6
More Than One Change per Statement........................5F7
Text Pointer Comparisons..................................5F8
Section 7: Invocation of User Filters and programs..........5G
Introduction..5G1
Programs Subsystem..5G2
Examples of User Programs 5G3

PART FOUR: Advanced L10 Programming............................6

Section 1: Executable Programs...............................6A
Introduction..6A1
Moving Around a File......................................6A2
Input/Output..6A3
Calling NLS Commands......................................6A4

Opening Files..6A5
Other Useful Procedures..............................6A6
Globals of Interest:.................................6A7
Section 2: Error Handling -- SIGNALs....................6B
Introduction...6B1
Trapping Signals.....................................6B2
Cancelling Signal Traps..............................6B3
Specific Signals.....................................6B4
Generating Signals...................................6B5

ASCII 7-BIT CHARACTER CODES.................................7

INTRODUCTION 2

NLS provides a variety of commands for file manipulation and viewing. Editing commands allow the user to insert and change the text in a file. Viewing commands (viewspecs) allow the user to control how the system prints or displays the file. Line truncation and control of statement numbers are examples of these viewing facilities. 2a

Occasionally one may need more sophisticated view controls than those available with the viewspec and viewchange features in NLS. 2b

For example, one may want to see only those statements that contain a particular word or phrase. 2b1

Or one might want to see one line of text that compacts the information found in several longer statements. 2b2

One might also wish to perform a series of routine editing operations without specifying each of the NLS commands over and over again. 2c

User-written programs may tailor the presentation of the information in a file to particular needs. Experienced users may write programs that edit files automatically. 2d

User-written programs currently must be coded in ARC's procedure-oriented programming language, L10. NLS itself is coded in L10. L10 is a high-level language which must be compiled into machine-readable instructions. 2e

This document describes three general types of programs:
--simple filters that control what is portrayed on the user's teletype or display (Parts One and Two),
--programs that may modify the statements as they decide whether to print them (Parts Two and Three),
--those that, like commands, are explicitly given control of the job and interact with the user (Part Four). 2f

User programs that control what material is portrayed take effect when NLS presents a sequence of statements in response to a command like Print (or Jump in DNLS). 2f1

In processing such a command, NLS looks at a sequence of statements, examining each statement to see if it satisfies the viewspecs then in force. At this point NLS may pass the statement to a user-written program to see if it satisfies the requirements specified in that program. If the user program returns a value of TRUE, the (passed) statement is printed and the next statement in the sequence is tested; if FALSE, NLS just goes on to the next statement. 2f1a

While the program is examining the statement to decide whether or not to print it, it may modify the contents of the statement. Such a program can do anything the user can do with NLS commands. 2f2

For more complicated tasks, control may be passed explicitly to the program. In this case, a user program appears as a special-purpose subsystem having (in addition to the supervisor commands) one or more commands. Once such a program is loaded, it can be used just like any of the standard subsystems. (The MESSAGE program is an example.) 2f3

This document describes the L10 programming language used at ARC. 2g

Part One is intended for the general user. 2g1

It is a primer on Content Analyzer Patterns. This does not involve learning the L10 language nor programming. This section can stand alone, and the general (if somewhat experienced) NLS user should find it useful. 2g1a

Part Two is intended for the beginning programmer. 2g2

It presents a hasty overview of L10 programming, with enough tools to write simple programs. This is intended as an introduction for the beginning L10 programmer, who we assume is reasonably familiar with NLS (its commands, subsystems, and capabilities) and has some aptitude for programming. 2g2a

Part Three is a more complete presentation of L10. 2g3

It is intended to acquaint a potential L10 programmer in enough of the language and NLS environment to satisfy most requirements for automated editing programs. Many of the concepts in Part Two are repeated in Part Three so that it may stand alone as an intermediate programmer's reference guide. 2g3a

Part Four presents more advanced L10 tools and an introduction to CML, allowing command syntax specification. 2g4

This should give the programmer the ability to write programs which work across files, which move through files in other than the standard sequential order, and which interact with the user. 2g4a

We suggest that those who are new to L10 begin with Section 1 and read this document one section at a time, pausing between sections to try out the concepts presented by actually writing patterns or programs that put the new ideas to experimental use. Hands-on experience is of at least as much value as this tutorial. If you have problems at any point, you should get help from ARC before proceeding to the next section. 2g5

More complete documentation can be found in (7052,1). For examples of user programs which serve a variety of needs, consult the User Programs Library Table of Contents (programs,-contents,1). For information about commands mentioned, ask for the programming subsystem with the NLS Help command. 2h

PART ONE: Content Analyzer Patterns 3

Section 1: Introduction 3a

Content analysis patterns cannot affect the format of a statement, nor can they edit a file. They can only determine whether a statement should be printed at all. They are, in a sense, a filter through which you may view the file. More complex tasks can be accomplished through programs, as described later in this document. 3a1

The Content Analyzer filter is created by typing in (or selecting from the text in a file) a string of a special form. This string is called the "Content Analyzer Pattern". Each statement is checked against the pattern before it is printed; only statements that are described by the pattern will be printed. 3a2

Some quick examples of Content Analyzer Patterns: 3a3

'($LD ') will show all statements whose first character is an open parenthesis, then any number of letters or digits, then a close parenthesis. 3a3a

["blap"] will show all statements with the string "blap" somewhere in them. 3a3b

SINCE (3-JUN-73 00:00) will show all statements edited since June 3, 1973 3a3c

The next part of this section will describe the elements which make up Content Analyzer Patterns, followed by some examples. The final subject of this section is how to put them to use. 3a4

Section 2: Patterns 3b

Elements of Content Analyzer Patterns 3b1

Content Analyzer Patterns describe certain things the system must check before printing a statement. It may check one or a series of things. The Content Analyzer searches a statement from the beginning, character by character, for described elements. As it encounters each element of the pattern, the Content Analyzer checks the statement for the occurrence of that pattern; if the test fails, the whole statement is failed (unless there was an "or" condition, as described later) and not printed; if the test is passed, an imaginary marker moves on to the next character in the statement, and the next test in the pattern is considered. 3b1a

The pattern may include any sequence of the following elements; the Content Analyzer moves the marker through the statement checking for each element of the pattern in turn: 3b1b

```
Literal Strings                                                  3b1c
   'c            the given character (e.g. a lower case c)
   "string"      the given string (may include
                 non-printing characters, such as spaces)
Character classes                                                3b1d
   CH            any character
   L             lowercase or uppercase letter
   D             digit
   UL            uppercase letter
   LL            lowercase letter
   ULD           uppercase letter, or digit
   LLD           lowercase letter, or digit
   LD            lowercase or uppercase letter, or digit
   NLD           not a letter nor digit
   PT            any printing character
   NP            any non-printing character (e.g. space)
Special characters                                               3b1e
   SP            a space
   TAB           tab character
   CR            a carriage return
   LF            line feed character
   EOL           TENEX EOL character
   ALT           altmode character
Special elements                                                 3b1f
   ENDCHR        beginning and end of every
                 statement; can't scan past it
```

TRUE is true without checking anything in statement

ID= id statement created by user whose ident is given

ID# id statement not created by user whose ident is given

BEFORE (d-t) statement edited before given date and time

SINCE (d-t) statement edited since given date and time

e.g. BEFORE (1 OCT 1974 00:00) ;

The date and time must both appear, in the parentheses. It accepts almost any reasonable date and time syntax.

Examples of valid dates:

17-APR-74	17 APRIL 74
APR-17-74	17/5/1974
APR 17 74	5/17/74
APRIL 17, 1974	

Examples of valid times:

1:12:13	1234:56
1234	1:56AM
1:56-EST	1200NOON
16:30 (4:30 PM)	
12:00:00AM (midnight)	
11:59:59AM-EST (late morning)	
12:00:01AM (early morning)	

Scan direction 3b1g

< set scan direction to the left

> set scan direction to the right

The default, re-initialized for each new statement, is scan to the right.

Combining Elements 3b2

These elements may be combined in any order. Spaces within the pattern are ignored (except in literal strings) so they may be used to make reading easier for you. Several operators can modify the elements: 3b2a

NUMBER -- multiple occurrences 3b2b

A number preceding any element other than one of the "Special elements" means that the test will succeed only if it finds exactly that many occurrences of the element. If there aren't that many, the statement will be rejected. Even though there may be more, it will stop after that many and go on to check the next element in the pattern.

3UL means three upper case letters

$ -- range of occurrences 3b2c

A dollar sign ($) preceding any element other than the "Special elements" means "any number of occurrences of". This may include zero occurrences.

$'- means any number of dashes

A number in front of the dollar sign sets a lower limit.
3$D means three or more digits

A number after the dollar sign sets an upper limit for the search. It will stop after that number and then check for the next element in the pattern, even if it could have found more.
$3LD means from zero to three letters or digits
5$7PT means from 5 to 7 (inclusive) printing characters

[] -- floating scan 3b2d

To do other than a character by character check, you may enclose an element or series of elements in square brackets []. The Content Analyzer will scan a statement until the element is found. (If the element is not in square brackets, the whole statement fails if the very next character or string fails the test of the next element.) This test will reject the statement if it can't find the element anywhere in the statement. If it succeeds, it will leave the marker for the next test just after the string satisfying the contents of the square brackets.

"start" means check to see if the statement begins with the string "start" (or, if it is in the middle of a pattern, check the next 5 characters to see if they are s t a r t).

["start"] means scan until it finds the string s t a r t.
[3D] means scan until it finds three digits.
[3D ':] means scan until it finds three digits followed by a colon

- -- negation 3b2e

If an element is preceded by a minus sign -, the statement will pass that test if the element does not occur.

-LD means anything other than a letter or digit, such as punctuation, invisibles, etc.

You may put together any number of any of these to form a pattern. 3b2f

e.g. 1SPT [",NLS;" 1SD] -SP

Logic in Patterns 3b3

More sophisticated patterns can by written by using the logic features of L10. Generally, an expression is executed left to right. The following operations are done in the given order:

()
/
NOT
AND
OR 3b3a

() 3b3b

Parentheses (and square brackets for floating scans) may be used to group elements. It is good practice to use parenthesis liberally.

/ 3b3c

/ means "either or"; the element will be true if either element is true.

(3D L / 4D) means either three digits and a letter or four digits.

Sometimes you may want want the scan to pass your marker over something if it happens to be there (an optional element). "TRUE" is true without testing the statement. If the other tests fail, the imaginary marker is not moved.

(D / TRUE) looks for a digit and passes the imaginary marker over it. If the next character is not a digit, it will just go on to the next test element in the pattern without moving the marker. This test always passes.

i.e. It is used to scan past something(s) which may or may not be there.

Since expressions are executed from left to right, it does no good to have TRUE as the first option. (If it is first, the test will immediately pass without trying to scan over any elements.)

NOT 3b3d

NOT will be TRUE if the element or group of elements enclosed in parentheses following the NOT is false.

NOT LD will pass if the next character is neither a letter nor a digit.

Since the slash is executed first, NOT D / 'h will be true if the next character is NEITHER a digit nor the letter "h". It is the same as NOT (D/'h).

AND 3b3e

AND means both of the two separated groups of elements must be true for the statement to pass.

SINCE (3/6/73 00:00) AND ID#NDM means statements written since March 6, 1973 by someone other than NDM.

OR 3b3f

OR means the test will be true if either of the separated elements is true. It does the same thing as slash, but after "AND" and "NOT" have been executed, allowing greater flexibility.

D AND LLD OR UL means the same as (D AND LLD) OR UL
D AND LLD / UL means the same as D AND (LLD / UL)

While such patterns are correct and succinct, parentheses make for much clearer patterns. Elements within parentheses are taken as a group; the group will be true only if the statement passes all the requirements of the group. It is a good idea to use parentheses whenever there might be any ambiguity.

Section 3: Examples of Content Analyzer Patterns 3c

D 2$LD / ["CA"] / ["Content Analyzer"] 3c1

This pattern will match any of three types of statements: those beginning with a numerical digit followed by at least two characters which may be either letters or digits, and statements with either the patterns "CA" or "Content Analyzer" anywhere in the statement. 3c1a

Note the use of the square brackets to permit a floating scan -- a search for a pattern anywhere in the statement. Note also the use of the slash for alternatives.

BEFORE (25-JAN-72 12:00) 3c2

This pattern will match those statements created or modified before noon on 25 January 1972. 3c2a

(ID = HGL) OR (ID = NDM) 3c3

This pattern will match all statements created or modified by users with the identifiers "HGL" or "NDM". 3c3a

[(2L (SP/TRUE) / 2D) D '- 4D] 3c4

This pattern will match characters in the form of phone numbers anywhere in a statement. Numbers matched may have an alphabetic exchange followed by an optional space (note the use of the TRUE construction to accomplish this) or a numerical exchange. 3c4a

Examples include DA 6-6200, DA6-6200, and 326-6200.

[ENDCHR] < "cba" 3c5

This will pass those statements ending with "abc". It will go to the end of the statement, change the scan direction to left, and check for the characters "cba". Note that since you are scanning backwards, to find "abc" you must look for "cba". Since the "cba" is not enclosed in square brackets, it must be the very last characters in the statement. 3c5a

Section 4: Using the Content Analyzer 3d

Content Analyzer Patterns may be entered in two ways: 3d1

1) From the BASE subsystem, use the command: 3d1a

Set Content (pattern) To PATTERN OK

2) From the PROGRAMS subsystem, use the command: 3d1b

Compile Content (pattern) PATTERN OK

OK means "Command Accept", a control-D or, in TNLS (by default) a carriage return.

In either case: 3d2

1) Patterns may be typed in from the keyboard, or 3d2a

2) they may be addressed from a file. 3d2b

In this case, the pattern will be read from the first character addressed and continue until it finds a semicolon (;) so you must put a semicolon at the end of the pattern (in the file).

Viewspec j must be on (i.e. Content Analyzer off) when entering a pattern. 3d2c

Entering a Content Analyzer Pattern automatically does two things: 3d3

1) compiles a small user program from the characters in the pattern, and 3d3a

2) takes that program and "institutes" it as the current Content Analyzer filter program, deinstituting any previous pattern. 3d3b

"Instituting" a program means selecting it as the one to take effect when the Content Analyzer is turned on. You may have more than one program compiled but only one instituted.

When a pattern is deinstituted, it still exists in your program buffer space and may be instituted again at any time with the command in the PROGRAMS subsystem:

Institute Program PROGRAM-NAME (as) Content (analyzer) OK

The programs may be refered to by number instead of name. They are numbered sequentially, the first entered being number 1.

All the programs you have compiled and the one you have instituted may be listed with the command in the PROGRAMS subsystem:

Show Status (of programs buffer) OK

Programs may build up in your program buffer. To clear the program buffer, use the PROGRAMS subsystem command:

Delete All (programs in buffer) OK

We recommend that you do this before each new pattern, unless you specifically want to preserve previous patterns.

To invoke the Content Analyzer: 3d4

When viewspec i is on, the instituted Content Analyzer program (if any) will check every statement before it is printed (or displayed). 3d4a

If a statement does not pass all of the requirements of the Content Analyzer program, it will not be printed.

In DNLS, if no statements from the top of the screen on pass the Content Analyzer, the word "Empty" will be displayed.

Note: You will not see the normal structure since one statement may pass the Content Analyzer although its source does not. Viewspec m (statement numbers on) will help you determine the position of the statement in the file.

When viewspec k is on, the instituted Content Analyzer search program will check until it finds one statement that passes the requirements of the pattern. Then, the rest of the output (branch, plex, display screen, etc.) will be printed without checking the Content Analyzer. 3d4b

When viewspec j is on, no Content Analyzer searching is done. This is the default state; every statement in the output (branch, plex, display screen, etc.) will be printed. Note that i, j, and k are mutually exclusive. 3d4c

Notes on the use of Content Analyzer filters: 3d5

Some NLS commands are always affected by the current viewspecs (including i,j, or k): 3d5a

Output

Jump (in DNLS)

Print (in TNLS)

Most NLS commands ignore the Content Analyzer in their editing. The following BASE subsystem commands offer the option of specifying viewspecs, or "Filters", (which may turn on the Content Analyzer) which apply only for the purpose of that one command and affect what statements the command works on: 3d5b

Copy

Delete

Move

Substitute

At this point, it would be wise to practice until you become proficient at Content Analyzer patterns. You might begin by trying to use some of the patterns given in the above examples, and then try writing a few patterns of your own. These patterns are both a useful NLS tool and a basic component of many L10 programs. 3d6

PART TWO: Introduction to L10 Programming 4

Section 1: Content Analyzer Programs 4a

Introduction 4a1

When you specify a Content Analyzer Pattern, the PROGRAMS subsystem constructs a program which looks for the pattern in each statement and only displays the statement if the pattern matching succeeds. You can gain more control and do more things if you build the program yourself. The program will be used just like the simple pattern program and has many of the same limitations. Programs are written in NLS just like any other text file. They then can be converted to executable code by a compiler. This code resides (or is loaded) in your programs buffer space; it can be instituted as the current Content Analyzer filter program like a Content Analyzer Pattern. 4a1a

Program Structure 4a2

If you specify a Content Analyzer Pattern, NLS compiles a small program that looks like this (with the word "pattern" standing for whatever you typed in): 4a2a

```
PROGRAM name

   (name) PROCEDURE;

      IF FIND pattern THEN RETURN(TRUE) ELSE RETURN(FALSE);

      END.

   FINISH
```

All L10 programs must begin with a header statement, the word PROGRAM (all caps) followed by the name of the first procedure to be executed (all lower-case). This name is also the name of the program. If the program is being compiled into a file (to be described at the end of this section), the word FILE should be substituted for the word PROGRAM. 4a2b

```
e.g.  PROGRAM first
      FILE deldir
```

(Note: the Content Analyzer makes up a program name consisting of UP#!xxxxx , where

is a sequential number, the first pattern being number one, and

xxxxx is the first five characters of your pattern.)

The body of a program consists of a series of DECLARATION statements and PROCEDURES (in any order). In the above case, the program consisted of only one small procedure and no declarations. When the program is loaded into your programs buffer space, the declarations reserve space in the system to store information (variables). When the program is used as a Content Analyzer filter program, the first procedure is called for each statement. It may in turn call other procedures and access variables in the program or in the NLS system. 4a2c

```
e.g.  DECLARE x, y, z   (described below)
      (first) PROCEDURE;
          ...
```

The end of the program is delimited by the word "FINISH" (in all upper case). 4a2d

Comments may be enclosed in percent signs (%) anywhere in the program, even in the middle of L10 statements. The L10 compiler will ignore them. 4a2e

Except within literal strings, variable names and special L10 words, spaces are ignored. It is good practice to use them liberally so that your program will be easy to read. Also, NLS file structure is ignored. Structure is, however, very valuable in making the program readable, and it is good practice to use it in close correlation to the program's logical structure. For instance, the programmer usually makes each of the elements of a program (declarations, procedures, and FINISH) seperate statements, below the header statement in file structure. This point will be discussed further later. 4a2f

So far, we have file which looks something like: 4a2g

```
PROGRAM name1

    DECLARE ... ;

    DECLARE ... ;
```

(name1) PROCEDURE ;

(name2) PROCEDURE ;

FINISH

Procedure Structure 4a3

Each procedure must begin with its header statement. This header statement is a name enclosed in parentheses followed by the word PROCEDURE, and terminated by a semicolon. 4a3a

e.g. (name) PROCEDURE ;

The body of the procedure may consist of Local declarations, then L10 statements. An L10 statement is any program instruction, terminated by a semicolon. The body must at some point return control to the procedure that called it. All this will be discussed more later. 4a3b

The procedure must end with the terminal statement: 4a3c

END.

Example: 4a4

PROGRAM compare 4a4a

```
% Content analyzer. Displays statement if first two
visibles are the same. %
DECLARE TEXT POINTER pt1, pt2, pt3, pt4;  %reserves
                             space for ("declares") four
                             text pointers named "pt1"
                             through "pt4"%
DECLARE STRING vis1[100], vis2[100];  %reserves 100
                             characters of space for each
                             of two string variables named
                             "vis1" and "vis2".%
(compare) PROCEDURE ;
   IF FIND SNP ^pt1 1$PT ^pt2 SNP ^pt3 1$PT ^pt4 THEN
                             %set pointers around first
                             two visibles (strings of
                             printng characters)%
      BEGIN                  %if it found two visibles%
      *vis1* _ pt1 pt2 ;     %put visibles in strings%
      *vis2* _ pt3 pt4 ;
      IF *vis1* = *vis2* THEN RETURN(TRUE);  %compare
                             contents of strings, return
                             and display the statement
                             if identical%
      END;
   RETURN (FALSE) ;          %otherwise, return and don't
                             display%
   END.
FINISH
```

Declaration Statements 4a5

As you may have guessed from the above example, Content Analyzer programs can deal with variables (like text pointers and strings), while patterns cannot. 4a5a

Text Pointers 4a5b

A text pointer points to a particular location within an NLS statement (or into a string, as described later).

The text pointer points between two characters in a statement. By putting the pointers between characters, a single pointer can be used to mark both the end of one string and the beginning of the string starting with the next character.

Text pointers are declared with the following Declaration statement:

```
DECLARE TEXT POINTER name ;
```

Strings 4a5c

String variables hold text. When they are declared, the maximum number of characters is set.

To declare a string:

```
DECLARE STRING name[num] ;
```

num is the maximum number of characters allowed for the string.

e.g. DECLARE STRING lstring[100];

declares a string named "lstring" with a maximum length of 100 characters and a current length of 0 characters (it's empty).

You can refer to the contents of a string variable by surrounding the name with asterisks.

e.g. *lstring* is the string stored in the variable named "lstring".

You can put the text between two text pointers in a string variable with the L10 statement:

```
*lstring* _ ptr1 ptr2 ;
```

where ptr1 and ptr2 are the names of previously declared and set text pointers, and lstring is a previously declared string variable.

These variables will retain their value from one statement to the next. Other types of variables and their use will be discussed in detail in Part Three, Section 3. 4a5d

Body of the Procedure 4a6

RETURN Statement 4a6a

No matter what it does, every procedure must return control to the procedure that called it. The statement which does this is the RETURN statement.

e.g. RETURN;

A RETURN statement may pass values to the procedure that called it. The values must be enclosed in parentheses after the word RETURN.

e.g. RETURN (1,23,47);

A Content Analyzer program must return either a value of TRUE or of FALSE. If it returns the value TRUE (1), the statement will be printed; if it returns FALSE (0), the statement will not be printed.

```
i.e. RETURN (TRUE);   will print the statement
     RETURN (FALSE);  will not print the statement
```

The RETURN statement often is at the end of a procedure, but it need not be. For example, in the middle of the procedure you may want to either RETURN or go on depending on the result of a test.

Other than the requirement of a RETURN statement, the body of the procedure is entirely a function of the purpose of the procedure. A few of the many possible statements will be described here; others will be introduced in Part Three of this document. 4a6b

FIND Statement 4a6c

One of the most useful statements for Content Analyzer programs is the FIND statement. The FIND statement specifies a Content Analyzer pattern to be tested against the statement, and text pointers to be manipulated and set, starting from the Current Character Position (that invisible marker refered to in Section 1). If the test succeeds, the character position is moved past the last character read. If the test fails, the character position is left at the position prior to the FIND statement and the values of all text pointers set within the statement will be reset.

FIND pattern ;

The Current Character position is initialized to BEFORE THE FIRST CHARACTER, and the scan direction is initialized to left to RIGHT, FOR EACH NEW STATEMENT passed to the Content Analyzer program.

Any simple Content Analyzer pattern (as describe above) is valid in a FIND statement. In addition, the following elements can be incorporated in the pattern:

stringname

the contents of the string variable

*ptr

store current scan position into the text pointer specified by ptr, the name of a declared text pointer

_NUM ptr

back up the specified text pointer by the specified number (NUM) of characters. If NUM is not specified, 1 will be assumed. Backup is in the direction opposite to the current scan direction.

ptr

Set current character position to this position. ptr is the name of a previously set text pointer.

SF(ptr)

The Current Character Position is set to the front of the statement in which the text pointer ptr is set and scan direction is set from left to right.

SE(ptr)

The Current Character Position is set to the end of the statement in which the text pointer ptr is set and scan direction is set from right to left.

BETWEEN ptr ptr (pattern)

Search limited to between positions specified. ptr is a previously set text pointer; the two must be in the same statement or string. Current Character Position is set to first position before the pattern is tested.

e.g. BETWEEN pt1 pt2 (2D [.] SNP)

FINDs may be used as expressions as well as free-standing statements. If used as an expression, for example in IF statements, it has the value TRUE if all pattern elements within it are true and the value FALSE if any one of the elements is false.

e.g. IF FIND pattern THEN ... ;

Complicated example:

```
IF FIND "sf $NP '( $(LD/'-) ') [".  " *str*] SE(sf) $NP
'. THEN RETURN(TRUE) ELSE RETURN(FALSE);
```

IF Statement 4a6d

IF causes execution of a statement if a tested expression is TRUE. If it is FALSE and the optional ELSE part is present, the statement following the ELSE is executed. Control then passes to the statement immediately following the IF statement.

```
IF testexp THEN statement ;

IF testexp THEN statement1 ELSE statement2 ;
```

The statements within the IF statement can be any valid L10 statement, but are not followed by the usual semicolon; the whole IF statement is treated like one statement and followed by the semicolon.

e.g.

```
IF FIND [5D] THEN RETURN(FALSE) ELSE RETURN(TRUE) ;
```

Programming Style: File Structure 4a7

You may remember that the compiler which converts your NLS text to code ignores file structure. This allows you to use structure to make your program text easier to read and understand. Logical use of structure often facilitates the actual programming task as well. Some conventions have developed at ARC in this respect. All of these should seem obvious and logical to you. 4a7a

All declarations and PROCEDURE statements should be one level below the PROGRAM statement.

All local declarations (not yet described) and code should be one level below the PROCEDURE statement.

It is good style, and makes for much easier programming, to list what you want to do as comment statements (in percent signs) at the level below the PROCEDURE statement. Then you can go back and fill in the code that accomplishes the task described in each comment statement. The code should go one level below the comment.

We will later describe how to block a series of statements where one is required. These blocks should go a level below the statement of which they are a part.

File structure should follow the logical structure of the program as closely as possible.

```
e.g.   IF FIND [SD]

            THEN RETURN(TRUE)

            ELSE RETURN(FALSE);
```

Using Content Analyzer Programs 4a8

Once the Content Analyzer program has been written (in an NLS file), there are two steps in using it. First, the program must be "compiled," i.e. translated into machine-readable code; the compiled code is "loaded" into a space reserved for user programs (the user programs buffer). Secondly, the loaded program must be "instituted" as the current Content Analyzer program. 4a8a

There are two ways to compile and load a program: 4a8b

1) You may compile a program and load it into your programs buffer all in one operation. The program header statement must have the word PROGRAM in it. When the user resets his job or logs off, the compiled code will disappear.

First, enter the Programs subsystem with the command:

Goto Programs OK

Then you may compile the program with the command:

Compile L10 (user program at) SOURCE OK

SOURCE is the address of the PROGRAM statement.

2) You may compile a program into a file and then load it into your buffer as a separate operation. The program can then be loaded from the file into your user programs buffer at any time without recompiling. The header statement must use the word FILE instead of PROGRAM. Use the PROGRAMS subsystem command:

Compile File (at) SOURCE (using) L10 (to file) FILENAME OK

The FILENAME must be the same as the program's name.

The code file is called a REL (RELocatable code) file. Whenever you wish to load the program code into the user programs buffer, use the PROGRAMS subsystem command:

Load REL (file) FILENAME OK

Once a compiled program has been loaded (by either route), it must be instituted. This is done with the PROGRAMS subsystem command: 4a8c

Institute Program PROGRAM-NAME
(as) Content (analyzer program) OK

The named program will be instituted as the current Content Analyzer program, and any previous program will be deinstituted (but will remain in the buffer).

Again, the programs in the buffer are numbered, the first in being number one. You may use the number instead of the program's name as a shorthand for PROGRAM-NAME.

To invoke the Content Analyzer using whatever program is currently instituted, use the viewspec i, j, or k, as described in Part One, Section 4 (3d4). 4a8d

Problems 4a9

Given these few constructs, you should now be able to write a number of useful Content Analyzer programs. Try programming the following: 4a9a

1) Show those statements which have a number somewhere in the first 20 characters.

2) Show those statements where the first visible in the statement is repeated somewhere in the statement.

Sample solutions: 4a9b

Problem 1

```
PROGRAM number
   DECLARE TEXT POINTER ptr1, ptr2 ;
   (number) PROCEDURE ;
      FIND ^ptr1 $20CH ^ptr2 ;
      IF FIND BETWEEN ptr1 ptr2 ( [D] )
         THEN RETURN(TRUE)
         ELSE RETURN(FALSE);
      END.
   FINISH
```

problem 2

```
PROGRAM vis
   DECLARE TEXT POINTER ptr1, ptr2 ;
   DECLARE STRING str[500] ;
   (vis) PROCEDURE ;
      FIND $NP ^ptr1 1$PT ^ptr2 ;
      *str* _ ptr1 ptr2 ;
      IF FIND ptr2 [NP *str* NP]
         THEN RETURN(TRUE)
         ELSE RETURN(FALSE);
      END.
   FINISH
```

Section 2: Content Analyzer Programs: Modifying Statements 4b

Introduction 4b1

Content Analyzer programs may edit the statements as well as decide whether or not they are printed. They are very useful where a series of editing operations has to be done time and time again. This section will introduce you to these capabilities. All these constructs wil be covered in detail in Part Three. 4b1a

A Content Analyzer program has several limitations. It can manipulate only one file and it can look at statements only in sequential order (as they appear in the file). It cannot back up and re-examine previous statements, nor can it skip ahead to other parts of the file. It cannot interact with the user. Part Four provides the tools to overcome these limitations. 4b1b

String Construction 4b2

Statements and the contents of string variables may be modified by either of the following two statements: 4b2a

ST ptr _ strlist ;

The whole statement in which the text pointer named "ptr" resides will be replaced by the string list (to be described in a minute).

ST ptr ptr _ strlist ;

The part of the statement from the first ptr to the second ptr will be replaced by the string list.

ptr may be a previously set text pointer or SF(ptr) or SE(ptr).

String variables may also be modified with the string assignment statement: 4b2b

stringname _ strlist ;

The string list (strlist) may be any series of string designators, seperated by commas. The string designators may be any of the following (other possibilities to be described later): 4b2c a string constant, e.g. "ABC" or 'w

ptr ptr

the text between two text pointers previously set in either a statement or a string

stringname

a string name in asterisks, refering to the contents of the string

E.g.: 4b2d

```
ST p1 p2 _ *string* ;
  or
ST p1 _ SF(p1) p1, string, p2 SE(p2);
```

(Note: these have exactly the same meaning.)

Example: 4b3

PROGRAM delsp 4b3a

```
% Content analyzer. Deletes all leading spaces from
statements. %
DECLARE TEXT POINTER pt; %reserves space for
                           ("declares") a text pointer
                           named "pt"%
(delsp) PROCEDURE ;
   IF FIND 1$SP ^pt THEN %scans over leading spaces,
                           then sets pointer%
      ST pt _ pt SE(pt); %replaces statement with text
                           from pointer to statement end%
   RETURN (FALSE) ;       %return, don't display anything%
   END.
FINISH
```

More Than One Change per Statement 4b4

Part of a text pointer is a character count. This count stays the same until the text pointer is again set (to some other position), even though the statement has been edited. If, for example, you have the statement 4b4a

abcdefghijklmnopqrstuvwxyz

and if you have set a pointer between the "d" and the "e", it will always point between the fourth and fifth characters in the statement. If you then delete the character "a", your pointer will be between the "e" and the "f", now the fourth and fifth characters. For this reason, you probably want to do a series of edits beginning with the last one in the statement and working backwards through the statement. 4b4b

Controlling Which Statements are Modified 4b5

In TNLS, the Content Analyzer program will be called for commands which construct a printout of the file (Print and Output). The program will run on every statement for which it is called (e.g. every statement in the branch during a Print Branch command) which pass all the other viewspecs. Once you have written, compiled, and instituted a program which does some editing operation, the Print command is the easiest way to run the program on a statement, branch, plex, or group. 4b5a

In DNLS, the system will call the Content Analyzer program whenever the display is recreated (e.g. viewspec f and the Jump commands), and also for the Output commands. If the program returns TRUE, it will only run on enough statements to fill the screen. It is safer to have programs that edit the file return FALSE. Then when you set viewspec i, it will run on all statements from the top of the display on, and when it is done it will display the word "Empty". At that point, change to viewspec j and recreate the display with viewspec f, then all statements including the changes will be displayed. You can control which statements are edited with level viewspecs and the branch only (g) or plex only (l) viewspecs. 4b5b

After having run your program on a file, you may wish to Update to permanently incorporate the changes in the file. It is wise to Update before you run the program so that, if the program does something unexpected, you can Delete Modifications and return to a good file. 4b5c

Problems 4b6

Try writing the following programs: 4b6a

1) Remove any invisibles from the end of each statement.

2) Make the first visible a statement name (surrounded by parenthesis) if it is a word (letters and digits).

Sample solutions: 4b6b

problem 1

```
PROGRAM endinv
   DECLARE TEXT POINTER ptr ;
   (endinv) PROCEDURE ;
      IF FIND ^ptr SE(ptr) 1SNP ^ptr
         THEN ST ptr _ SF(ptr) ptr ;
      RETURN (FALSE) ;
      END.
   FINISH
```

problem 2

```
PROGRAM makename
   DECLARE TEXT POINTER ptr1, ptr2 ;
   (makename) PROCEDURE ;
      IF FIND SNP ^ptr1 1SLD ^ptr2 NP
         THEN ST ptr1 _ '(, ptr1 ptr2, '), ptr2 SE(ptr2);
      RETURN(FALSE)
      END.
   FINISH
```

PART THREE: Basic L10 Programming

From here on has not been updated; the NLS commands mentioned may be syntactically incorrect, and the section on user interface is obsolete (having been replaced by CML). New documentation should be expected by the end of the year.

5

Section 1: The User Program Environment 5a

Introduction 5a1

User-written Content Analyzer programs run in the framework of the portrayal generator. They may be invoked in several ways, described below, whenever one asks to view a portion of the file, e.g., with a Print command in TNLS, with any of the output commands, and with the Jump command in DNLS. 5a1a

All of the portrayal generators in NLS have at least two sections -- the formatter and the sequence generator; if the user invokes a Content Analyzer program of his own, the portrayal generator will have one additional part -- the user program. 5a1b

Executable programs are independent of the portrayal generator, although they are welcome to make use of it. They are called as procedures by the Programs subsystem, and have all the powers of any other NLS procedure. 5a1c

Sequence Generator 5a2

The sequence generator looks at statements one at a time, beginning at the point specified by the user. It observes viewspecs like level truncation in determining which statements to pass on to the formatter. 5a2a

For example, the viewspecs may indicate that only the first line of statements in the two highest levels are to be output. The default NLS sequence generator will return pointers only to those statements passing the structural filters; the formatter will further truncate the text to only the first line.

When the sequence generator finds a statement that passes all the viewspec requirements, it returns the statement to the formatter and waits to be called again for the next statement in the sequence. 5a2b

One of the viewspecs that the sequence generator pays particular attention to is "i" -- the viewspec that indicates whether a user filter is to be applied to the statement. If this viewspec is on, the sequence generator passes control to a user Content Analyzer program, which looks at the statement and decides whether it should be included in the sequence. If the statement passes the Content Analyzer (i.e. the user program returns a value of TRUE), the sequence generator sends the statement to the formatter; otherwise, it processes the next statement in the sequence and sends it to the user Content Analyzer program for verification. (The particular user program chosen as a filter is determined by what program is Instituted as the current Content Analyzer program, as described below.) 5a2c

Formatter 5a3

The formatter section arranges text passed to it by the sequence generator in the style specified by other viewspecs. The formatter observes viewspecs such as line truncation, length and indenting; it also formats the text in accord with the requirements of the output device. 5a3a

The formatter works by calling the sequence generator, formatting the text returned, then repeating this process until the sequence generator decides that the sequence has been exhausted (e.g. the branch has been printed) or the formatter has filled the desired area (e.g. the display screen). 5a3b

Content Analyzers 5a4

The NLS Portrayal Generator, made up of the formatter, the sequence generator, and user filters, is invoked whenever the user requests a new "view" of the file, for example through the use of the TNLS "Print" command or any of the output commands. Thus if one had a user Content Analyzer program compiled, instituted, and invoked, one could have a printout made containing only those statements in the file satisfying the pattern. 5a4a

When a user writes an content analyzer filter program, the main routine must RETURN to the Portrayal Generator. The RETURN must have an argument which is checked by the sequence generator. If the value of that argument is TRUE, the statement will be passed to the formatter to be displayed or printed; if the value is FALSE, it will not be displayed. In DNLS, if you display any statements, the program will stop after filling the screen. If you are not displaying any statements, the program will run on either the whole file, a plex (viewspec l), or a branch (viewspec g). These along with level clipping viewspecs give one precise control over what statements in the file will be passed to the program. 5a4b

User-Written Sequence Generators 5a5

A user may provide his own sequence generator to be used in lieu of the regular NLS sequence generator. Such a program may call the normal NLS sequence generator, as well as content analysis filters and Executable L10 programs. It may even call other user-written sequence generators. 5a5a

This technique provides the most powerful means for a user to reformat (and even create) files and to affect their portrayal. However, since writing them requires a detailed knowledge of the entire NLS program code, the practice is limited to experienced NLS programmers, and will not be covered in this document. However, the information provided in these next sections should provide you with enough to accomplish most any task. 5a5b

Section 2: Program Structure 5b

An NLS user program consists of the following elements, which must be arranged in a definite manner with strict adherence to syntactic punctuation: 5b1

The header - 5b1a

a statement consisting of the word PROGRAM, followed by the name of a procedure in the program. Program execution will begin with a call to the procedure with this name.

```
PROGRAM name
```

The word FILE should be substituted for the word PROGRAM if the code is to be compiled into a file to be saved.

The body - 5b1b

consists of declarations and procedures in any order:

1) declaration statements which specify information about the data to be processed by the procedures in the program and enter the data identifiers in the program's symbol table, terminated by a semicolon.

```
e.g.  DECLARE x,y,z ;
      DECLARE STRING test[500] ;
      REF x, z;
```

Declaration statements will be covered in Section 3 (4c).

2) Procedures which specify certain execution tasks. Each procedure must consist of -

the procedure name enclosed in parentheses followed by the word PROCEDURE and optionally an argument list containing names of variables that are passed by the calling procedure for referencing within the called procedure. This statement must be terminated by a semicolon.

```
e.g.  (name) PROCEDURE ;
      (name) PROCEDURE (param1, param2) ;
```

the body of the procedure which may consist of LOCAL, REF, and L10 statements.

LOCAL and REF declarations within a procedure must precede executable code. They will be covered in Section 3 (4c).

L10 statements will be covered in Sections 4 and 5 (4d) (4e).

the statement that terminates the procedure (note the final period):

END.

The program terminal statement - 5b1c

FINISH

Comments may be enclosed in percent signs (%) anywhere in the program, even in the middle of L10 statements. They will be ignored. 5b1d

Except for within literal strings, variable names, and special L10 reserved words, spaces are ignored. It is good practice to use them liberally so that your program will be easy to read. Also, NLS file structure is ignored. Structure is, however, very valuable in making the program readable, and it is good practice to use it in close correlation to the program's logical structure. 5b1e

An example of a simple L10 program is provided here. The reader should easily understand this program after having studied this document. 5b2

```
PROGRAM delsp                                                          5b2a
   % Content analyzer. Deletes all leading spaces from
   statements. %
   DECLARE TEXT POINTER pt; %reserves space for
                             ("declares") a text
                             pointer named "pt"%
   (delsp) PROCEDURE ;
      IF FIND 1$SP ^pt THEN %scans over leading spaces,
                             then sets pointer%
         ST pt _ pt SE(pt); %replaces statement holding
                             pt with text from pointer
                             to statement end%
      RETURN (FALSE) ;      %return, don't display%
      END.
   FINISH
```

Section 3: Declarations 5c

Introduction 5c1

L10 declarations provide information to the compiler about the data that is to be accessed; they are not executed. Every variable used in the program must be declared somewhere in the system (either in your program or in the NLS system program). 5c1a

There are various types of declarations available; the most frequently used are discussed here. (Complete documentation is available in the L10 Reference Guide -- 7052.) 5c1b

Variables 5c2

Five types of variables are described in this document: simple, arrays, text pointers, strings, and referenced. Each can be declared on two levels: global or local. 5c2a

Global Variables 5c2b

A global variable is represented by an identifier and refers to a cell in memory which is known and accessible throughout the program. Global variables are defined in the program's DECLARE statements or in the NLS system program.

Variables specified in these declarations are outside any procedure and may be used by all procedures in the program. Many globals are defined as part of the NLS system; user programs have complete access to these. Be very careful about changing their values, however.

Local Variables 5c2c

A local variable is known and accessible only to the procedure in which it appears. Local variables must appear in a procedure argument list or be declared in a prodecure's LOCAL declaration statements (to be explained below). Any LOCAL declarations must precede the executable statements in a procedure.

Local variables in the different procedures may have the same name without conflict. A global variable may not be declared as a local variable and a procedure name may be used as neither. In such cases the name is considered to be multiply defined and an error results.

Simple Variables 5c3

Simple variables represent one computer word, or 36 bits, of memory. Each bit is either on or off, allowing binary numbers to be stored in words. Each word can hold up to five ASCII 7-bit characters, a single number, or may be divided into fields and hold more than one number. 5c3a

Declaring a variable allocates a word in the computer to hold the contents of the variable. The variable name refers to the contents of that word. One may refer to the address of that computer word by preceding the variable name by a dollar sign ($).

For example, if one has declared a simple variable called "num", one may put the number three in that variable with the statement:

```
num _ 3 ;
```

One may add two to a variable with the statement:

```
num _ num + 2 ;
```

One may put the address of num into a variable called addr with the statement:

```
addr _ $num ;
```

One may refer to predefined fields in any variable by following the name of the variable with a period, then the field name. For example, the fields RH and LH are globally defined to be the right and left half of the word respectively; e.g.

```
num.LH _ 2 ;
num.RH _ 3 ;
```

Fields may be defined by the user with RECORD statements (not explained in this document). Additionally, you may refer to system-defined fields (e.g. RH). They divide words into fields by numbers of bits, so they may refer to any declared word. For example, the field "LH" refers to the left-most 18 bits in any 36-bit word.

Declaring Simple Global Variables 5c3b

```
DECLARE name ;
```

"name" is the name of the variable. It must be all lower-case letters or digits, and must begin with a letter.

e.g. DECLARE x1 ;

Optionally, the user may specify the initial value of the variable being declared. If a simple variable is not initialized at the program level, for safety it should be initialized in the first executed procedure in which it appears.

```
DECLARE name = constant ;
```

constant is the initial value of name. It may be any of the following:

- a numeric constant optionally preceded by a minus sign (-)

- a string, up to five characters, enclosed in quotation marks

- another variable name, causing the latter's address to be used as the value of name

Examples:

```
DECLARE x2 = 5 ;     %x2 contains the value 5%
DECLARE x3 = "OUT"; %x3 contains the word OUT%
DECLARE xx = x1;     %xx contains the address of x1%
```

Arrays 5c4

Multi-word (one-dimensional) array variables may be declared; computer words within them may be accessed by indexing the variable name. The index follows the variable name, and is enclosed in square brackets []. The first word of the array need not be indexed. The index of the first word is zero, so if we have declared a ten element array named "blah": 5c4a

```
blah    is the first word of the array
blah[1] is the second word of the array
blah[9] is the last word of the array
```

Declaring Global Array Variables 5c4b

```
DECLARE name[num] ;
```

num is the number of elements in the array if the array is not being initialized. It must, of course, be a integer.

e.g. DECLARE sam[10];

declares an array named "sam" containing 10 elements.

Optionally, the user may specify the initial value of each element of the array. If array values are not initialized at the program level, for safety they should be initialized in the first executed procedure in which the array is used.

DECLARE name = (num, num, ...) ;

num is the initial value of each element of the array. The number of constants implicitly defines the number of elements in the array. They may be any of the constants allowed for simple variables.

Note: there is a one-to-one correspondence between the first constant and the first element, the second constant and the second element, etc.

Examples:

DECLARE numbs=(1,2,3);

declares an array named numbs containing 3 elements which are initialized such that:

numbs = 1
numbs[1] = 2
numbs[2] = 3

DECLARE motley=(10,$blah);

declares an array named motley containing 2 elements which are initialized such that:

motley = 10

motley[1] = $blah
= the address of the variable "blah"

Text Pointers 5c5

A text pointer is an L10 feature used in string manipulation constructions. It is a two-word entity which provides information for pointing to particular locations within text, whether in free standing strings or an NLS statement. 5c5a

The text pointer points between two characters in a statement or string. By putting the pointers between characters a single pointer can be used to mark both the end of one substring and the beginning of the substring starting with the next character, thereby simplifying the string manipulation algorithms and the way one thinks about strings.

A text pointer consists of a string identifier and a character count. 5c5b

The first word, called an stid, contains three system-defined fields:

```
stfile -- the file number (if an NLS statement)
stastr -- a bit indicating string, not an NLS statement
stpsid -- the psid of the statement;
          every statement has a unique number (psid)
          attached to it.
```

The stid is the basic handle on a statement in L10.

The second word contains a character count, with the first position being 1 (before the first character).

For example, one might have the following series of assignment statements which fill the three fields of the first word and the second word with data, with pt being the name of a declared text pointer:

```
pt.stfile _ fileno;  %fileno a simple variable
                     with a number in it%
pt.stastr _ FALSE;   %a statement, not a string%
pt.stpsid _ origin;  %all origin statements have the
                     psid = 2; origin is a global
                     variable with the value 2 in it%
pt[1] _ 1;           %the word one after pt (i.e. the
                     character count) gets 1, the
                     beginning of the statement%
```

It is important that stid's be initialized properly to avoid strange errors.

Declaring Text Pointers 5c5c

DECLARE TEXT POINTER pt ;

The names p1, p2, p3, p4, and p5 are globally declared and reserved for system use.

Strings 5c6

String variables are a series of words holding text. When they are declared, the maximum number of characters is set. The first word contains the two globally defined fields: 5c6a

M -- the maximum number of characters the string can hold
L -- the actual number of characters currently in the string

The next series of words (as many as are required by the maximum string size) hold the actual characters, five per word, in ASCII 7-bit code. 5c6b

str refers to the contents of the string variable "str".

str refers to the first word of the string variable "str".

str.M refers to the maximum declared length of the string variable "str" (an integer).

str.L refers to the current length of the string stored in the string variable "str" (an integer).

Declaring Strings 5c6c

The DECLARE STRING enables the user to declare a global string variable by initializing the string and/or declaring its maximum character length.

To declare a string:

DECLARE STRING name[num] ;

num is the maximum number of characters allowed for the string

e.g. DECLARE STRING lstring[100];

declares a string named "lstring" with a maximum length of 100 characters and a current length of 0 characters

To declare and initialize a string:

DECLARE STRING name="Any string of text" ;

The length of the literal string defines the maximum length of the string variable.

e.g. DECLARE STRING message="RED ALERT";

declares the string message, with an actual and maximum length of 9 characters and contains the text "RED ALERT"

Referenced Variables 5c7

Reference Declarations 5c7a

After a simple variable has been declared, the REF statement can define it to be a pointer to some other variable. A referenced variable holds the address of another declared variable of any type. Whenever the referenced variable is mentioned, L10 will operate on the other variable instead, as if it were declared in that procedure and named at that point.

This is useful when you wish a procedure to know about a multi-word variable. In procedure calls, you are only allowed to pass one-word parameters. If you wish a called procedure to know about a text pointer, array, or string, you may pass the address of the multi-word variable. Then, in the called procedure, you must REF the formal parameter receiving that address. From then on in the called procedure, when you refer to the parameter, you are actually operating on the multi-word variable declared in some other procedure to which the local REFed variable points.

Example:

If the simple variable "loc" in the current procedure has been REFed and contains the address of the string "str" local to the calling procedure, then operations on loc actually operate on the string in str:

```
*mes* _ *loc*;        %mes gets the string in
                       str%
```

```
*loc* _ "corpuscle";  %str gets the string
                       "corpuscle"%
```

Similarly, you cannot return multi-word variables from a called procedure. If you wish a procedure to return a string, you must declare the string as a local in the CALLING procedure, pass its address to a REFed variable in the called procedure, and then you can modify the string as if it were local to the called procedure (and return nothing).

Unreferenced Variables 5c7b

One may refer to the actual contents (an address) of a referenced variable (i.e. "unref" it) by preceding the referenced variable name with an ampersand (&). If, for example, an address was passed to a REFed local, and you wish now to pass that address on to another procedure, you can unref it.

e.g. if x has been REFed and holds the address of y:

```
z _ x ;  %z gets the CONTENTS of y%
z _ &x;  %z gets the ADDRESS of y%
```

This construct might be used, for example, if one procedure has been passed the address of a string, operates on it, then wishes to pass (the address of) that string on to another procedure that it calls.

REFing Simple Variables 5c7c

Once a simple variable has been declared, it may be REFed with the statement:

```
REF var ;
```

It will be a reference from then on in that procedure, and you must always use the ampersand to refer to the actual contents of the variable.

Declaring Many Variables in One Statement 5c8

One may avoid putting several individual declarations of variables in a series by putting variables of similar type, initialized or not, in a list in one statement following a single DECLARE, separated by commas and terminated by the usual semicolon. Array and simple varibles may be put together in one statement. 5c8a

Examples:

```
DECLARE x, y[10], z = (1, 2, -5);
DECLARE TEXT POINTER tp, sf, pt1, pt2 ;
DECLARE STRING lstring[100], message="RED ALERT" ;
```

Declaring Locals 5c9

Program level declarations (DECLARE and REF) and procedures may appear in any order. However, procedure level declarations (LOCAL and REF inside a procedure) must appear before any executable statements in the procedure. The different types of variables may be declared in any order, but a variable must be declared before it can be REFed. 5c9a

With one exception, a local variable declaration statement is just the same as a global with the word "LOCAL" substituted for the word "DECLARE". The one exception is that LOCAL declarations can not initialize the variables. 5c9b

Examples:

```
LOCAL var, flag, level[12] ;
LOCAL TEXT POINTER tp, pt, sf ;
LOCAL STRING test[100], out[2000] ;
```

When a procedure is called by another procedure, the calling procedure may pass one-word parameters. The procedure receives these values in simple local variables declared in the PROCEDURE statement's parameter list. For example, two locals will automatically be declared and set to the passed values whenever the procedure "procname" is called: 5c9c

```
(procname) PROCEDURE (var1, var2) ;
```

var1 and var2 must not be declared again in a LOCAL statement. They may, however, be REFed by a REF statement, as discussed above, and used throughout the procedure.

The statement which calls procname may look like:

```
procname (locvar, 2) ;
```

var1 will be initialized to the value of the variable "locvar" and var2 will get the value 2.

Section 4: Statements 5d

Introduction 5d1

This section will describe some of the types of statements with which one can build a procedure. The term "expression" (often abbreviated to "exp") will be used in this section, and will be explained in detail in Section 5 (4e). 5d1a

Assignment 5d2

In the assignment statement, the expression on the right side of the "_" is evaluated and stored in the variable on the left side of the statement. 5d2a

```
var _ exp ;
```

where var = any global, local, referenced or unreferenced variable.

One may make a series of assignments in one statement by enclosing the list of variables and the list of expressions in parentheses. The order of evaluation of the expressions is left to right. The expressions are evaluated and pressed onto a stack; after all are evaluated they are popped from the stack and stored in the variables. 5d2b

```
(var1, var2, ...) _ (exp1, exp2, ...) ;
```

Naturally, the number of expressions must equal the number of variables.

Example:

```
(a, b) _ (c+d, a-b)
```

The expression c+d is evaluated and stacked, the expression a-b is evaluated and stacked, the value of a-b is popped from the stack and stored into b, and finally, the value of c+d is popped and stored into a. It is equivalent to:

```
temp1 _ c+d ;
temp2 _ a-b ;
b _ temp2 ;
a _ temp1 ;
```

One may assign a single value to a series of variables by stringing the assignments together: 5d2c

```
var1 _ var2 _ var3 _ exp ;
```

var1, var2, and var3 will all be given the value of the expression.

Example:

```
a _ b _ 0;
```

Both a and b will be given the value zero. This type of statement can be useful in initializing a series of variables at the beginning of a procedure.

IF Statement 5d3

This form causes execution of a statement if a tested expression is TRUE. If the expression is FALSE and the optional ELSE part is present, the statement following the ELSE is executed. Control then passes to the statement immediately following the IF statement. 5d3a

```
IF testexp THEN statement ;

IF testexp THEN statement1 ELSE statement2 ;
```

The statements within the IF statement can be any statement, but are not followed by the usual semicolon; the whole IF statement is treated like one statement and followed by the semicolon. 5d3b

e.g. 5d3c

```
IF y=z THEN y_y+1 ELSE y_z ;
```

CASE Statement 5d4

This form is similar to the IF statement except that it causes one of a series of statements to be executed depending on the result of a series of tests. 5d4a

```
CASE testexp OF
   relop exp : statement ;
   relop exp : statement ;
   relop exp : statement ;
        .
        .
   ENDCASE statement ;
```

where relop = any relational operator (>=, <, =, IN, etc.) see Section 5 (4e3).

The CASE statement provides a means of executing one statement out of many. The expression after the word "CASE" is evaluated and the result left in a register. This is used as the left-hand side of the binary relations at the beginning of the various cases. Each expression is evaluated and compared according to the relational operator to the CASE expression. If the relationship is TRUE, the statement is executed. If the relationship is FALSE, the next expression and relatonal operator will be tried. If none of the relations is satisfied, the statement following the word "ENDCASE" will be executed. Control then passes to the statement following the CASE statement 5d4b

Note that the relop and expressions are followed by a colon, and the statements are terminated with the usual semicolon. The word ENDCASE is not followed by a colon. In ENDCASE, the statement may be left out -- this is the equivalent of having a NULL statement there; nothing will happen.

Example:

```
CASE c OF
   = a: x _ y;                     %Executed if c = a%
   > b: (x, y) _ (x+y, x-y);       %Executed if c > b%
   ENDCASE y _ x;                  %Executed otherwise%

CASE char OF
   = D: char _ '1;    %if char = the code for a digit%
   = UL: char _ '0;   %if char = the code for an
                       upper-case letter%
   ENDCASE;           %otherwise nothing%
```

Several relations may be listed at the start of a single case; they should be separated by commas. The statement will be executed if any of the relations is satisfied. 5d4c

```
CASE testexp OF
   relop exp : statement ;
   relop exp, relop exp : statement ;
   relop exp, relop exp, relop exp : statement ;
      .
      .
   ENDCASE statement ;
```

Example:

```
CASE c OF
   =a, <d:                        %Executed if c=a or c<d%
      x _ y;
   >b, =d:                        %Executed if c>b or c=d%
      (x,y) _ (x+y,x-y);
   ENDCASE                        %Executed otherwise%
      y _ x;
```

As a point of style, the conditions of the CASE statement should be put one level below the CASE statement in the source (text) file. The statements (if they are more than one line) may be put one level below the condition. 5d4d

LOOP Statement 5d5

The statement following the word "LOOP" is repeatedly executed until control leaves by means of some transfer instruction within the loop. 5d5a

```
LOOP statement;
```

where statement = any executable L10 statement

Example:

```
LOOP IF a>=b THEN EXIT LOOP ELSE a _ a+1 ;
```

It is assumed that a and b have been initialized before entering the loop.

The EXIT construction is described below. It is extremely important to carefully provide for exiting a loop.

WHILE...DO Statement 5d6

This statement causes a statement to be repeatedly executed as long as the expression immediately following the word WHILE has a logical value of TRUE or control has not been passed out of the DO loop by EXIT CASE (described below). 5d6a

```
WHILE exp DO statement ;
```

exp is evaluated and if TRUE the statement following the word DO is executed; exp is then reevaluated and the statement continually executed until exp is FALSE. Then control will pass to the next statement. 5d6b

For example, if you want to fill out a string with spaces through the 20th character position, you could:

```
WHILE str.L < 20 DO *str* _ *str*, SP;   %what's already
                                          there, then a space%
```

Remember that the first word of every string variable has two globally defined fields:

L -- actual length of contents of string variable
M -- maximum length of string variable

UNTIL...DO Statement 5d7

This statement is similar to the WHILE...DO statement except that statement following the DO is executed until exp is TRUE. As long as exp has a logical value of FALSE the statement will be executed repeatedly. 5d7a

```
UNTIL exp DO statement ;
```

Example:

```
UNTIL a>b DO a _ a+1 ;
```

DO...UNTIL/DO...WHILE Statement 5d8

These statements are like the preceding statements, except that the logical test is made after the statement has been executed rather than before. 5d8a

```
DO statement UNTIL exp;

DO statement WHILE exp;
```

Thus the specified statement is always executed at least once (the first time, before the test is made). 5d8b

FOR...DO Statement 5d9

The FOR statement causes the repeated execution of the statement following "DO" until a specific terminal value is reached. 5d9a

```
FOR var UP UNTIL relop exp DO statement;

   (UP will be assumed if left out.)

FOR var DOWN UNTIL relop exp DO statement;
```

where

var = the variable whose value is incremented or decremented each time the FOR statement is executed

relop = any relational operator (described in 4e3c)

exp = when combined with relop, determines whether or not another iteration of the FOR statement will be performed.

e.g. `FOR i UP UNTIL > 7 DO a _ a + t[i] ;` 5d9b

Optionally, the user may initialize the variable and may increment it by other than the default of one. 5d9c

```
FOR var _ exp1 UP   exp2 UNTIL relop exp3 DO statement;
               DOWN
```

where

exp1 = an optional initial value for var. If exp1 is not specified, the current value of var is used.

exp2 = an optional value by which var will be incremented (if UP specified) or decremented (if DOWN specified). If exp2 is not specified, a value of one will be assumed.

Note that exp2 and exp3 are recomputed on each iteration.

Example:

```
FOR k _ n UP k/2 UNTIL > m*3 DO x[k] _ k;
```

is equivalent to

```
k _ n;
LOOP
   BEGIN
   IF k >m*3 THEN EXIT LOOP;
   x[k] _ k;
   k _ k + k/2;
   END;
```

BEGIN...END Statement 5d10

The BEGIN...END construction enables the user to group several statements into one syntactic statement entity. A BEGIN...END construction of any length is valid where one statement is required. 5d10a

BEGIN statement ; statement ; ... END ;

Example:

```
IF a >= b*c THEN
   BEGIN
   a_b;
   c_d+5;
   END
ELSE
   BEGIN
   a_c;
   b_d+2;
   c_b*d*7
   END ;
```

Note the use of NLS file structure to clarify the logic and seperate the blocks. Blocks should always be put one level below the statement of which they are a part.

EXIT Statement 5d11

This construction provides for forward branches out of CASE or iterative statements. The optional number (num) specifies the number of lexical levels of CASE or iterative statements respectively that are to be exited (if loops are nested within loops). If a number is not given then 1 is assumed. All of the iterative statements (LOOP, WHILE, UNTIL, DO, FOR) can be exited by the EXIT LOOP construct. A CASE statement can be left with an EXIT CASE instruction. EXIT and EXIT LOOP have the same meaning. 5d11a

```
EXIT LOOP num  or  EXIT num
EXIT CASE num

   where num is an optional integer.

Examples:

   LOOP
      BEGIN
      ........
      IF test THEN EXIT;
         %the EXIT will branch out of the LOOP%
      ........
      END;

   UNTIL something DO
      BEGIN
      ........
      WHILE test1 DO
         BEGIN
         ........
         IF test2 THEN EXIT;
            %the EXIT will branch out of the WHILE%
         ........
         END;
      ........
      END;

   UNTIL something DO
      BEGIN
      ........
      WHILE test1 DO
         BEGIN
         ........
         IF test2 THEN EXIT 2;
            %the EXIT 2 will branch out of the UNTIL%
         ........
         END;
      ........
      END;

   CASE exp OF
      =something:
```

```
        BEGIN
        ........
        IF test THEN EXIT CASE;
           %the EXIT will branch out of the CASE%
        ........
        END;
     ........
```

REPEAT Statement 5d12

This construction provides for backward branches to the front of CASE or iterative statements. The optional number has the same meaning as in the EXIT statement. REPEAT and REPEAT CASE have the same meaning. 5d12a

```
REPEAT LOOP num

REPEAT CASE num (exp)   or   REPEAT num (exp)
```

If an expression is given with the REPEAT CASE, then it is evaluated and used in place of the expression given at the head of the specified CASE statement. If the expression is not given, then the one at the head of the CASE statement is reevaluated. 5d12b

Examples: 5d12c

```
CASE exp1 OF
   =something:
      BEGIN
      ........
      IF test1 THEN REPEAT;
         %REPEAT with a reevaluated exp1%
      ........
      IF test2 THEN REPEAT(exp2);
         %REPEAT with exp2%
      ........
      END;
   ........
   ENDCASE ;

LOOP
   BEGIN
   ........
   IF test THEN REPEAT LOOP;
      %REPEAT LOOP will go to the top of the LOOP%
   ........
   END;
```

DIVIDE Statement 5d13

The divide statement permits both the quotient and remainder of an integer division to be saved. The syntax for the divide statement is as follows: 5d13a

```
DIV exp1 / exp2 , quotient , remainder ;
```

Quotient and remainder are variable names in which the respective values will be saved after the division. 5d13b

e.g.

```
DIV a / b, a, r ;
```

a will be set to a/b to the greatest integer with r getting the remainder

PROCEDURE CALL Statement 5d14

This statement is used to direct program control to the procedure specified. 5d14a

```
procname (exp, exp, ... : var, var, ...) ;
```

Where procname = the name of a procedure 5d14b

exp = any valid L10 expression (explained in Section 5 -- 4e). The set of expressions separated by commas is the argument list for the procedure. 5d14c

var = any variable. The set of variables is used to store the results of the procedure if there is more than one result. 5d14d

The argument list consists of a number of expressions separated by commas. The number of arguments should equal the number of formal parameters for the procedure. The argument expressions are evaluated in order from left to right. Each expression (parameter) must evaluate to a one-word value. To pass an array, text pointer, string, or any multi-word parameter, the programmer may pass the address of the first word of the variable, then REF the receiving local in the called procedure. 5d14e

For example, one may pass an stid directly, but to pass a text pointer, you must pass the address of the text pointer and REF the receiving parameter.

The procedure may return one or more values. The first value is returned as the value of the procedure call. Therefore, if only one value is returned, one might say: 5d14f

```
a _ proc (b) ;
```

In this context, the procedure call is an expression.

If more than one value is returned by the called procedure, one must specify a list of variables in which to store them. The list of variables for multiple results is separated from the list of argument expressions by a colon. The number of locations for results need not equal the number of results actually returned. If there are more locations than results, then the extra locations get an undefined value. If there are more results than locations, the extra results are simply lost. The first RETURN value is still taken only as the value of the procedure call. 5d14g

Example:

If procedure "proc" ends with the statement

```
RETURN (a,b,c)
```

then the statement

```
q _ proc(:r,s);
```

results in (q,r,s) _ (a,b,c).

A procedure call may just exist as a statement alone without returning a value. Not all procedures require parameters, but the parentheses are mandatory in order to distinguish a procedure call from other constructs. 5d14h

e.g. af();

If a block of instructions are used repeatedly, or are duplicated in different sections of a program, it is often wise to make them a separate procedure and simply call the procedure when appropriate. 5d14i

A great many procedures are part of the NLS system and are available to your programs. A list of them is available in the file (nls,sysgd,). They should be used with care. SYSGD lists links to the source code, so that you can examine the procedure in detail to see just what it expects as arguments and what it returns. 5d14j

RETURN Statement 5d15

This statement causes a procedure to return control to the procedure which called it. Optionally, it may pass the calling procedure an arbitrary number of results. The order of evaluation of results is from left to right. 5d15a

```
RETURN ;

RETURN (exp, exp, ...) ;

E.g.  RETURN (TRUE, a+b) ;
      RETURN ( getnmf(stid) ) ;
```

GOTO Statement 5d16

Any statement may be labeled; one puts the desired label (a string of lower case letters and digits) in parentheses and followed by a colon at the beginning of a statement. 5d16a

```
(label): statement ;

e.g.  (there): a _ b + c ;
```

GOTO provides for unconditional transfer of control to a new location. 5d16b

```
GOTO label ;

e.g.  GOTO there ;
```

GOTO statements make debugging difficult and are not considered good style; they can usually be eliminated by use of procedure calls and the iterative statements. (Section 8 will mention the only condition in which they are necessary.) 5d16c

NULL Statement 5d17

The NULL statement may be used as a convenience to the programmer. It does nothing. 5d17a

```
NULL ;
```

Example:

```
CASE exp OF
   =0, =1: NULL;
   ENDCASE y_1;
```

Section 5: Expressions 5e

Introduction 5e1

This section will describe the composition of the expressions, which are an integral part of many of the statements described in the last section. 5e1a

Primitives 5e2

Primitives are the basic units which are used as the operands of L10 expressions. There are many types of elements that can be used as L10 primitives; each type returns a value which is used in the evaluation of an expression. 5e2a

Each of the following is a valid primitive: 5e2b

a constant (see below)

any valid variable name, refering to the contents (of the first word, if not indexed) of that variable

the contents of a string variable, refered to as *var*

a dollar sign ($) followed by a variable name, refering to the address of the variable

a procedure call which returns at least one value

the first (leftmost) value returned is the value of the procedure call; other values may be stored in other variables as described in Section 4 (4d14f).

an assignment (see below)

classes of characters; described in Sections 1 of Part One (3a2a3)

MIN (exp, exp, ...) the minimum of the expressions

MAX (exp, exp, ...) the maximum of the expressions

TRUE has the value 1

FALSE has the vaue 0

VALUE (astring) given the address of a string containing a decimal number, has the value of the number

READC (see below)

CCPOS (see below)

FIND

used to test text patterns and load text pointers for use in string construction (see Section 6 -- 4f3); returns the value TRUE or FALSE depending on whether or not all the string tests within it succeed.

POS

POS textpointer1 relop textpointer2

may be used to compare two text pointers. If the POS construction is not used, only the first words of the pointers (the stid's) will be compared. If a pointer is before another, it is considered less than the other pointer.

```
e.g.  POS pt1 = pt2
      POS first >= last
```

Constants 5e2c

A constant may be either a number or a literal constant.

There are several ways in which numeric values may be represented. A sequence of digits alone (or followed by a D) is interpreted as base ten. If followed by a B then it is interpreted as base eight. A scale factor may be given after the B for octal numbers or after a D for decimal numbers. The scale factor is equivalent to adding that many zeros to the original number.

Examples:

64 = 100B = 1B2

144B = 100 = 1D2

Literals may be used as constants as they are represented internally by numeric values. The following are valid literal constants:

-any single character preceded by an apostrophe

e.g. 'a represents the code for 141B.

-any string of up to five characters enclosed in quotation marks

e.g. "aa" represents the code for 141141B

-the following synonyms for commonly used characters:

ENDCHR -endcharacter as returned by READC

SP -space

ALT -Tenex's version of altmode or escape (=33B)

CR -carriage return

LF -line feed

EOL -Tenex EOL character

TAB -tab

BC -backspace character

BW -backspace word

C. -center dot

CA -Command Accept

CD -Command Delete;

Assignments 5e2d

An assignment can be used as a primitive in an expression.

The form a _ b has the effect of storing b into a and has the value of b as its value.

Another form of the assignment statement is:

a := b

This will store b into a, but have the old value of a as the value of the assignment when used as a primitive in an expression.

For example,

```
b _ (a := b) ;
```

The value of b will be put in a. The assignment will get the old value of a, which is then put in b. This transposes the values of a and b.

READC - ENDCHR 5e2e

The primitive READC is a special construction for reading characters from NLS statements or strings.

A character is read from the current character position in the scan direction set by the last CCPOS statement or string analysis FIND statement or expression. CCPOS and FIND are explained in detail in Section 6 of this document (4f2) and (4f3).

Attempts to read off the end of a string in either direction result in a special "endcharacter" being returned and the character position not being moved. This endcharacter is included in the set of characters for which system mneumonics are provided and may be referenced by the identifier "ENDCHR".

For example, to sequentially process the characters of a string;

```
CCPOS *str*;

UNTIL (char _ READC) = ENDCHR DO process(char);
```

(Note: READC may also be used as a statement if it is desired to read and simply discard a character).

CCPOS 5e2f

When used as a primitive, CCPOS has as its value the index of the character to the right of the current character position. If str = "glarp", then after CCPOS *str*, the value of CCPOS is 1 and after CCPOS SE(*str*) the value of CCPOS is 6 (one greater than the length of the string).

CCPOS is more commonly used as a statement to set the current character position for use in text pattern matching. This is discussed in detail in Section 6 below (4f2).

CCPOS may be useful as an index to sequentially process the first n characters of a string (assumed to have at least n characters)

Example:

```
CCPOS *str*;    %CCPOS now has the index value of
                one, the front of the string%
UNTIL CCPOS > n DO process(READC).
                %READC reads the next character
                and increments CCPOS%
```

Operators 5e3

Primitives may be combined with operators to form expressions. Four types of operators will be described here: arithmetic, relational, interval, and logical. 5e3a

Arithmetic Operators 5e3b

Operator	Meaning
unary +	positive value (when in front of a number)
unary -	negative value
+	addition
-	subtraction
*	multiplication
/	integer division (remainder not saved)
MOD	a MOD b gives the remainder of a / b
.V	a .V b = bit pattern which has 1's wherever either an a or b had a 1 and 0 elsewhere.
.X	a .X b = bit pattern which has 1's wherever either an a had a 1 and b had a 0, or a had a 0 and b had a 1, and 0 elsewhere.

.A	a .A b = bit pattern which has 1's wherever both a and b had 1's, and 0 elsewhere.

Relational Operators 5e3c

A relational operator is used in an expression to compare one quantity with another. The expression is evaluated for a logical value. If true, its value is 1; if false, its value is 0.

Operator	Meaning	Example	
=	equal to	4+1 = 3+2	(true, =1)
#	not equal to	6#8	(true, =1)
<	less than	6<8	(true, =1)
<=	less than or equal to	8<=6	(false, =0)
>	greater than	3>8	(false, =0)
>=	greater than or equal to	8>=6	(true, =1)
NOT other-relational-operator		6 NOT > 8	(true, =1)

Interval Operators 5e3d

The interval operators permit one to check whether the value of a primitive falls in or out of a particular interval.

IN (primitive, primitive) IN [primitive, primitive]

OUT (primitive, primitive) %equivalent to NOT IN%

The value is tested to see whether or not it lies within (or outside of) a particular interval. Each side of the interval may be "open" or "closed". Thus the values which determine the boundaries may be included in the interval (by using a square bracket) or excluded (by using parentheses).

Example:

x IN [1,100)

is the same as

(x >=1) AND (x < 100)

Logical Operators 5e3e

Every numeric value also has a logical value. A numeric value not equal to zero has a logical value of TRUE; a numeric value equal to zero has a logical value of FALSE.

```
Operator        Evaluation
--------        ----------

  OR            a OR b = TRUE  if a = TRUE  or  b = TRUE
                       = FALSE if a = FALSE and b = FALSE

  AND          a AND b = TRUE  if a = TRUE  and b = TRUE
                       = FALSE if a = FALSE or  b = FALSE

  NOT            NOT a = TRUE  if a = FALSE
                       = FALSE if a = TRUE
```

Expressions 5e4

Introduction 5e4a

An expression is any constant, variable, special expression form, or combination of these joined by operators and parentheses as necessary to denote the order in which operations are to be performed.

Special L10 expressions are: the FIND expression which is used for string manipulation, and the conditional IF and CASE expressions which may be used to give alternative values to expressions depending on tests made in the expressions. Expressions are used where the syntax requires a value. While certain of these forms are similar syntactically to L10 statements, when used as an expression they always have values.

Order of Operator Execution-- Binding Precedence 5e4b

The order of performing individual operations within an equation is determined by the heirarchy of operator execution (or binding precedence) and the use of parentheses.

Operations of the same heirarchy are performed from left to right in an expression. Operations in parentheses are performed before operations not in parentheses.

The order of execution of operators (from first to last) is as follows:

unary -, unary +

.A

.V, .X

*, /, MOD

+, -

relational tests (e.g., >=, <=, >, <, =, #, IN, OUT)

NOT relational tests (e.g., NOT >)

NOT

AND

OR

Conditional Expressions 5e4c

The two conditional constructs (IF and CASE) can be used as expressions as well as statements. As expressions, they must return a value.

IF Expressions

IF testexp THEN exp1 ELSE exp2

testexp is tested for its logical value. If testexp is TRUE then exp1 will be evaluated. If it is FALSE, then exp2 is evaluated.

Therefore, the result of this entire expression is EITHER the result of exp1 of exp2.

Example:

```
y _ IF x IN[1,3] THEN x ELSE 4;
                %if x = 1, 2, or 3, y_x; otherwise y_4%
```

CASE Expression

This form is similar to the above except that it causes any one of a series of expressions to be evaluated and used as the result of the entire expression.

```
CASE testexp OF
   relop exp : exp ;
   relop exp : exp ;
   relop exp : exp ;
       .
       .
   ENDCASE exp ;
```

where relop = any relational operator (>=, <, =, IN, etc. See above -- 4e3c)

In the above, the testexp is evaluated and used with the operator relops and their respective exps to test for a value of TRUE or FALSE. If TRUE in any instance, the companion expression to the right of the colon is executed and taken to be the value of the whole expression. A value of FALSE for all tests causes the next relop in the CASE expression to be tested against the testexp. If all relops are FALSE, the ENDCASE expression is taken to be the value of the whole expression.

Note that ENDCASE cannot be null; it must have a value.

As with the CASE statement, any number of cases may be specified, and each case may incude more than one relop and expression, seperated by commas.

Example:

```
y _ CASE x OF
   <3: x+1;
   =3, =4: x+2;
   =5: x;
   ENDCASE x*2;
```

Value of X	Value of y
2	3
3	5
4	6
5	5
6	12

String Expressions 5e4d

L10 also provides several expression forms which are used for string manipulation and evaluation. These are discussed in Section 6 of this document. When using string manipulation statement forms as expressions, parentheses may be necessary to prevent ambiguities.

Section 6: String Test and Manipulation 5f

Introduction 5f1

This section describes statements which allow complex string analysis and construction. The three basic elements of string manipulation discussed here are the Current Character Position (CCPOS) and text pointers which allow the user to delimit substrings within a string (or statement), patterns that cause the system to search the string for specific occurrences of text and set up pointers to various textual elements, and actual string construction. 5f1a

Current Character Position (CCPOS) 5f2

The Current Character Position is similar to the TNLS CM (Control Marker) in that it specifies the location in the string at which subsequent operations are to begin. All L10 string tests start their search from the current character position. In Content Analyzer programs, it is initialized to the BEGINNING OF EACH NEW STATEMENT. For each new statement, the scan direction is initialized to left to right. It is moved through the statement or through strings by FIND expressions. It may be set to a particular position in a statement or string by the L10 statement: 5f2a

```
CCPOS pos ;
      or
CCPOS *stringname*[exp] ;
```

pos is a position in a statement or string that may be expressed as any of the following: 5f2b

A previously declared and set text pointer.

If a text pointer is given after CCPOS, then the character position is set to that location. A text pointer points between two characters in a string. The scan direction over the text will remain unchanged.

e.g. CCPOS pt1 ;

String Front -- left of the first character

SF(stspec)

When SF is specified scanning will take place from left to right within the string.

stspec is a string specification that may be expressed as an stid (e.g. the first word of a previouly declared text pointer) or previously declared string name enclosed in asterisks.

Examples:

```
CCPOS SF(pt1) ;   %pt1 is a text pointer%
CCPOS SF(stid) ;  %stid is an stid%
CCPOS SF(*str*) ; %str is a string%
```

String End -- right of the last character

SE(stspec)

When SE is specified scanning will take place from right to left within the string.

If a string (*stringname*) is given after CCPOS, then the position is moved to that string. The scan direction is set left to right. 5f2c

Indexing the stringname (by specifying [exp]) simply specifies a particular position within the string. Thus *str*[3] puts the Current Character Position between the second and third characters of the string "str". If the scan direction is left to right, then the third character will be read next. If the direction is right to left, then the second will be read next.

```
e.g. CCPOS *str*[3] ;
```

If no indexing is given, then the position is set to the left of the first character in the string. This is equivalent to an index of 1.

```
e.g. CCPOS *str* ;
         means the same as
     CCPOS SF(*str*);
```

FIND Statement 5f3

The FIND statement specifies a string pattern to be tested against a statement or string variable, and text pointers to be manipulated and set, starting from the current character position. If the test succeeds the character position is moved past the last character read. If the test fails the character position is left at the position prior to the FIND statement and the values of all text pointers set within the statement will be reset. 5f3a

FIND pattern ;

FINDs may be used as expressions as well as free-standing elements. If used as an expression, for example in IF statements, it has the value TRUE if all pattern elements within it are true and the value FALSE if any one of the elements is false. 5f3b

e.g. IF FIND pattern THEN ... ;

FIND Patterns 5f4

A string pattern may be any valid combination of the following logical operators, testing arguments, and other non-testing parameters: 5f4a

Pattern Matching Arguments-- 5f4b

(each of these can be TRUE or FALSE)

string constant, e.g. "ABC"

or any character, preceded by an apostrophy

It should be noted that if the scan direction is set right to left the pattern string constant pattern should be reversed. In the above example, one would have to search for "CBA".

Any of the system defined mnemonics, as described in the last section (4e2c), such as "SP" or "CR", are also valid.

character class

look for a character of a specific class; if found, = TRUE, otherwise FALSE.

Character classes:

CH - any character
L - lowercase or uppercase letter

UL - uppercase letter
LL - lowercase letter
D - digit
LD - lowercase or uppercase letter or digit
NLD - not a letter or digit
ULD - uppercase letter or digit
LLD - lowercase letter or digit
PT - printing character
NP - nonprinting character

Example:

char = LD

is TRUE if the variable char contains a value which is a letter or a digit.

(elements)

look for an occurrence of the pattern specified by the elements. If found, = TRUE, otherwise FALSE. Elements may be any pattern; the parentheses serve to group the elements so as to be treated as a single element in any of the following elements.

-element

TRUE only if the element following the dash does not occur.

[elements]

TRUE if the pattern specified by the elements can be found anywhere in the remainder of the string. elements may be any pattern; the squarebrackets also group the elements so as to be treated as a single element. It first searches from current position. If the search failed, then the current position is incremented by one and the pattern is tried again. Incrementing and searching continues until the end of the string. The value of the search is FALSE if the testing string entity is not matched before the end of the string is reached.

NUM element

find (exactly) the specified number of occurrences of the element.

e.g. 3LD means three letters or digits

NUM1 $ NUM2 element

Tests for a range of occurrences of the element specified. If the element is found at least NUM1 times and at most NUM2 times, the value of the test is TRUE.

Either number is optional. The default value for NUM1 is zero. The default value for NUM2 is 10000. Thus a construction of the form "$3 CH" would search for any number of characters (including zero) up to and including three.

Examples:

2$4 UL - from two to four upper-case letters

$10 SP - up to ten spaces

1$ '. - one or more periods

ID = user-ident
ID # user-ident

if the string being tested is the text of an NLS statement then NIC ident of the user who created or last edited the statement is tested by this construction.

SINCE datim

if the string being tested is the text of an NLS statement, this test is TRUE if the statement was created or modified after the date and time (datim, see below) specified.

BEFORE datim

if the string being tested is the text of an NLS statement, this test is TRUE if the statement was created or modified before the date and time (datim, see below) specified.

Format of date and time for pattern matching

Acceptable dates and times follow the forms permitted by the TENEX system's IDTIM JSYS described in detail in the JSYS manual. It accepts "most any reasonable date and time syntax."

Examples of valid dates:

```
17-APR-70          APR-17-70
APR 17 70          17 APRIL 70
17/5/1970          5/17/70
APRIL 17, 1970
```

Examples of valid times:

```
1:12:13            1234
1234:56            1:56AM
1:56-EST           1200NOON
16:30  (4:30 PM)
12:00:00AM   (midnight)
11:59:59AM-EST   (late morning)
12:00:01AM   (early morning)
```

Examples:

```
BEFORE (MAR 19, 73 16:49)
SINCE (25-JUL-73 00:00)
```

These may not appear in Content Analysis patterns, but are valid elements in FIND statements in any program:

stringname

the contents of the string variable

BETWEEN pos pos (element)

Search limited to between positions specified. pos is a previously set text pointer; the two must be in the same statement or string. Scan character position is set to first position before the pattern is tested.

e.g. BETWEEN pt1 pt2 (2D [.] $NP)

Logical Operators-- 5f4c

These combine and delimit groups of patterns. Each compound group is considered to be a single pattern with the value TRUE or FALSE. If text pointers are set within a test pattern and the pattern is not TRUE, the values of those text pointers are reset to the values they had before the test was made. (See examples below.)

OR
AND
NOT
/

Other Elements-- 5f4d

These do not involve tests; rather, they involve some execution action. They are always TRUE for the purposes of pattern matching tests.

These may appear in simple Content Analysis Patterns:

<

set scan direction to the left

In this case, care should be taken to specify patterns in reverse, that is in the order which the computer will scan the text.

>

set scan direction to the right

TRUE

has no effect; it is generally used at the end of OR when a value of TRUE is desired even if all tests fail.

ENDCHR

Attempts to read off the end of a string in either direction result in a special "endcharacter" being returned and the character position is not moved. This endcharacter is included in the set of characters for which system mneumonics are provided and may be referenced by the identifier "ENDCHR".

These may not appear in simple Content Analysis Patterns:

pos

pos is a previously set text pointer, or an SE(pos) or

SF(pos) construction. Set current character position to this position. If the SE pointer is used, set scan direction from right to left. If the SF pointer is used, set scan direction from left to right.

e.g. FIND x; %sets CCPOS to position of previously set text pointer x%

_ ID

store current scan position into the textpointer specified by the identifier

_ [NUM] ID

back up the specified text pointer by the specified number (NUM) of characters. Default value for NUM is one. Backup is in the opposite direction of the current scan direction.

String Construction 5f5

One may modify an NLS statement or a string with the statement: 5f5a

ST pos _ strlist ;

The whole statement or string in which pos resides will be replaced by the string list.

ST pos pos _ strlist ;

The part of the statement or string from the first pos to the second pos will be replaced by the string list. "pos" may be a previously set text pointer or the SF(pos)/SE(pos) construction.

There are two additional ways of modifying the contents of a string variable: 5f5b

```
ST *stringname*[exp TO exp] _ strlist ;
        means the same as
*stringname*[exp TO exp] _ strlist ;
```

The string from the first position to the second position will be replaced by the string list. The square-bracketed range is entirely optional; if it is left off, the whole string will be replaced.

Note that the "ST" is optional when assigning a strlist to the contents of a string variable. The statement then resembles any simple assignment statement.

The string list (strlist) may be any series of string designators, seperated by commas. The string designators may be any of the following: 5f5c

the word NULL

represents a zero length (empty) string

string constant, e.g. "ABC" or 'W

part of any string or statement, denoted either by

two text pointers previously set in either a statement or a string

pos pos

a string name in asterisks, refering to the whole string

stringname

a string name in asterisks followed by an index, refering to a character in the string

stringname[exp]

(The index of the first character is one.)

a string name in asterisks followed by two indices, refering to a substring of the string

stringname[exp TO exp]

A construction of the form *str*[i TO j] refers to the substring starting with the ith character in the string up and including the jth character.

Examples:

str[7 TO 10] is the four character substring starting with the 7th character of str.

str[i TO str.L] is the string str without the first i-1 characters. (i is a declared variable.)

+ substring

substring capitalized

- substring

substring in lower case

exp

value of a general L10 expression taken as a character; i.e., the character with the ASCII code value equivalent to the value of the expression

STRING (exp1, exp2);

gives a string which represents the value of the expression exp1 as a signed decimal number. If the second expression is present, a number of that base is produced instead of a decimal number.

e.g. STRING (3*2) is the same as the string "6.0"

Examples: 5f5d

```
ST p1 p2 _ *string*;
  does the same as
ST p1 _ SF(P1) p1, *string*, p2 SE(p2);
```

assuming p1 and p2 have been set somewhere in the same statement. The latter reads "replace the statement holding p1 with the text from the beginning of the statement to p1, the contents of string, then the text from p2 to the end of the statement."

```
*st*[low TO high] _ "string";
  does the same as
*st* _ *st*[1 TO low-1], "string", *st*[high+1 TO st.L];
```

assuming low and high are declared simple variables.

Example: 5f6

Let a "word" be defined as an arbitrary number of letters and digits. The two statements in this example delete the word pointed to by the text pointer "t", and if there is a space on the right of the word, it is also deleted. Otherwise, if there is space on the left of the word it is deleted. 5f6a

The text pointers x and y are used to delimit the left and right respectively of the string to be deleted. 5f6b

```
IF (FIND t < sLD ^x > sLD (SP ^y / ^y x < (SP ^x / TRUE)) )
THEN
   ST x y _ NULL;
```
5f6c

The reader should work through this example until it is clear that it really behaves as advertised. 5f6d

More Than One Change per Statement 5f7

The second word of a text pointer, the character count, stays the same until the text pointer is again set to some other position (as does the first word), even though the statement has been edited. If, for example, you have the statement 5f7a

abcdefghijklmnopqrstuvwxyz

and if you have set a pointer between the "d" and the "e", it will always point between the fourth and fifth characters in the statement; the second word of the text pointer holds the number 5. If you then delete the character "a", your pointer will be between the "e" and the "f". For this reason, you probably want to do a series of edits beginning with the last one in the statement and working backwards. 5f7b

Text Pointer Comparisons 5f8

This may be used to compare two text pointers. 5f8a

```
POS pt1 =  pt2;
        #
        >
        <
        >=
        <=
```

pt1 and pt2 are a text pointers.

NOT may precede any of the relational operators. If the pointers refer to different statements then all relations between them are FALSE except "not equal" which is written # or NOT=. If the pointers refer to the same statement, then the truth of the relation is decided on the basis of their location within the statement.

A pointer closer to the front of the statement is "less than" a pointer closer to the end.

Section 7: Invocation of User Filters and Programs 5g

Introduction 5g1

The user-written filters described in this document may be imposed through the NLS command "Goto Programs". 5g1a

User sequence generator programs for more complex editing among many files may be written. Additionally, programs may be written in this L10 subset to be used to generate sort keys in the NLS Sort and Merge commands. Descriptions of these more complicated types of user programs and of NLS procedures which may be accessed by such programs is deferred until a later document. In such examples, however, the user would still make use of the commands in the NLS "Goto Programs" subsystem.

These NLS commands are used to compile, institute and execute User Programs and filters. 5g1b

Compilation--

is the process by which a set of instructions in a program is translated from the L10 language written in an NLS file into a form which the computer can use to execute those instructions.

Institution--

is the process by which a compiled Content Analyzer program is linked into the NLS running system for use as a filter.

Execution--

is the process in which control is passed to a compiled Executable program.

This section additionally presents examples of the use of the L10 programming language. These programs were written by members of ARC who are not experienced programmers. They do not make use of any constructions not explained in this manual. 5g1c

Programs Subsystem 5g2

Introduction 5g2a

This NLS subsystem provides several facilities for the processing of user written programs and filters. It is entered by using the NLS command:

Goto Programs OK

This subsystem enables the user to compile L10 user programs as well as Content Analyzer patterns, control how these are arranged internally for different uses, define how programs are used, and interrogate the status of user programs.

Programs subsystem commands 5g2b

After entering the Programs subsystem, the system expects one of the following commands:

Show Status of programs buffer

This command prints out information concerning active user programs and filters which have been compiled and/or instituted:

Show Status (of programs buffer) OK

When this command is executed the system will print:

-- the names of all the programs in the stack, including those generated for simple Content Analysis patterns, starting at the bottom of the stack. This stack contains the symbolic names of all compiled programs and a pointer to the corresponding compiled code. The stack is arranged in order of compilation with the first program compiled at the bottom of the stack.

-- the remaining free space in the buffer. The buffer contains the compiled code for all the current compiled programs. New compiled code is inserted at the first free location in this buffer.

-- the current Content Analyser Program or "None"

-- the current user Sequence Generator program or "None"

-- the user Sort Key program or "None"

Compile

L10 Program

This command compiles the program specified.

Compile L10 (user program at) ADDRESS OK

ADDRESS is the address of the first statement of the program.

This command causes the program specified to be compiled into the user program buffer and its name entered into the stack. The program is not instituted.

The name of the program is the visible following the word PROGRAM in the statement indicated by ADDRESS.

The program may be instituted or executed by the appropriate commands.

File

The user program buffer is cleared whenever the user resets or logs out of the system. If you have a long program which will be used periodically, you may wish to save the compiled code in a file which can be retrieved with the Load Program command. The command to compile into a file is:

Compile File (at) ADDRESS (using) L10 OK (to file) FILENAME OK

The FILENAME must be the same as the program name. The program will then be compiled and stored in the file of the given name (with the extension REL, unless otherwise specified). The user may then load it at any time.

Before doing this, the programmer must replace the word PROGRAM at the head of the file with the word FILE.

Content Analyzer Pattern

This command allows the user to specify a Content Analyzer pattern as a Content Analyzer filter.

Compile Content (analyzer filter) ADDRESS OK

The pattern must begin with the first visible after the SELECTON address, or at that point you may type it in. It will read the pattern up to a semicolon, so be sure to insert a semicolon where you want it to stop.

When this command is executed, the pattern specified is compiled into the buffer, its name is put on the stack, AND it is automatically instituted as the Content Analyzer filter.

Load Program

A pre-compiled program existing as a REL file may be loaded into the program buffer with the command:

Load Program FILENAME OK

If the FILENAME is specified without specifying an extension name, this command will search the connected directory, then the <user-progs> directory, for the following extensions:

```
REL  it will simply load the REL file
CA   it will load the program and institute it
     as the current content analyzer program
SK   it will load the program and institute it
     as the current sort key extractor program
SG   it will load the program and institute it
     as the current sequence generator program
```

Sort key extractor and sequence generator programs are more complex and are generally limited to experienced L10 programmers. Some are available in the User Programs Library (user-progs,-contents,1).

Delete

All

This command clears all programs from the user program area. All programs are deinstituted, the stack is cleared, and the buffer is marked as empty.

Delete All (programs in buffer) OK

Last

This command deletes the top (or most recent) program on the stack. The program is deinstituted if instituted, its name removed from the stack, and its space in the buffer marked as free.

Delete Last (Program in buffer) OK

Run Program

This command transfers control to the specified program.

```
Run Program PROGNAME OK
            NUMBER
```

PROGNAME is the name of a program which had been previously compiled. That is, PROGNAME must be in the buffer when this command is executed.

Instead of PROGNAME, the user may specify the program to be instituted by its number. This first program loaded into the buffer is number one.

Institute Program

This command enables the user to designate a program as the current Content Analyzer, Sequence Generator, or Sort Key extractor program.

```
Institute Program PROGNAME OK
                  NUM
 (as) CA (content analyzer) OK
      Content (analyzer) OK
      Sort (key extractor) OK
      Sequence (generator) OK
```

If a program has already been instituted in that capacity, it will be deinstituted (but not removed from the buffer and stack).

Instead of PROGNAME the user may specify the program to be instituted by number. The first program loaded into the buffer is number one.

Deinstitute Program

This command deactivates the indicated program, but does not remove it from the stack and buffer. It may be reinstituted at any time.

```
Deinstitute Content (analyzer program)   OK
            Sort (Key extractor program)
            Sequence (generator program)
```

Set Buffer size

The user programs buffer shares memory with data pages for files which the user has open, therefore increasing the size of the user programs buffer decreases the amount of space available for file data with a possible slowdown in response for that user. The initial size is set to 4 pages. This may be increased with the command:

Set Buffer (size) NUMBER OK

where NUMBER is the number of pages (512 words each) to be allocated to the user programs buffer.

If you get an "Error in Loading" message when attempting to compile a program or load a REL file, try increasing the buffer size.

You may reset the buffer size (to four pages) with the command:

Reset Buffer (size) OK

Assemble File

Files written in Tree-Meta can be assembled directly from the NLS source file with the Assemble File command. This aspect of NLS programming will not be described in this document.

Examples of User Programs 5g3

The following are examples of user programs which selectively edit statements in an NLS file on the basis of text searched for by the pattern matching capabilities. Examples of more sophisticated user programs, including sort keys, can be found in the <user-progs> directory through the file (user-progs,-contents,). One can find out how the standard NLS commands work by tracing them through, beginning with (nls, syntax, 2). A table of contents to all the global NLS routines available to the user can be found in (nls, sysgd, 1). 5g3a

Example 1 -- Content Analyzer program 5g3b

```
PROGRAM outname % removes the text and delimiters () of NLS
statement names from the beginning of each statement %
   DECLARE TEXT POINTER sf;
   (outname)PROCEDURE;
      IF FIND SNP '( ['')] ^sf THEN %found and set
                                    pointer after name%
         BEGIN
         ST sf _ sf SE(sf);
         RETURN(TRUE);
         END
      ELSE RETURN(FALSE);
      END.
   FINISH
```

Example 2 -- Content Analyzer program 5g3c

```
PROGRAM changed   %This program checks to see if a statement
was written after a certain date. If it was, the string
"[CHANGED]" will be put at the front of the statement. %
   (changed)PROCEDURE;
      LOCAL TEXT POINTER pt;
      %remember, CCPOS is initialized to the beginning of
      each new statement%
      IF FIND ^pt SINCE (25-JAN-72 12:00) THEN
         ST pt pt _ "[CHANGED]"; %the substring of zero
                                  length is replaced with
                                  "[CHANGED]"%
      RETURN(FALSE);
      END.
   FINISH
```

Example 3 -- Executable program 5g3d

```
FILE toc %This program will generate a table of contents
branch with statement numbers  %
   (toc) PROCEDURE ;
      % declarations %
         LOCAL level, da, vspec, last, place ;
```

```
         LOCAL TEXT POINTER ptr ;
         LOCAL STRING num[5] ;
         REF da ;
         num.L _ ptr _ 0;  %initialization%
      % input file and number of levels %
         IF nlmode=typewriter
            THEN
               BEGIN
               crlf() ;
               typeas($"Table of Contents generator:  Select
               file ");
               tbug ($ptr) ;  %get a bug from the tty%
               crlf() ;
               typeas ($"Number of levels of depth:  ") ;
               txtlit ($num) ;  %get a text string from the
               tty%
               crlf() ;
               typeas($"running... ");
               END
            ELSE %display%
               BEGIN
               dn($"") ;  %clear the name register%
               DSP (< Table of Contents - Select file) ;
               INPUT STID ptr CA;
               DSP (_< Levels of depth) ;
               INPUT NUMBER num CA ;
               DSP (< Table of Contents being generated) ;
               END;
      % set to origin %
         ptr.stpsid _ origin ;
         ptr[1] _ 1 ;
         level _ VALUE ($num); %evaluate number string%
         level _ MIN (50, MAX (1,level)); %levels of depth%
      % insert table of contents statement %
         ptr _ cis (ptr, $"Table of Contents", down);
         %command insert statement procedure%
      % get viewspec words %
         &da _ lda(); %get address of display area records,
         which hold all information about display window,
         e.g. viewspecs%
         vspec _ da.davspec ; %copy viewspec word%
         vspec.vslev _ level ;  %adjust level viewspec%
         vspec.vsbrof _ vspec.vsplxf _ FALSE;  %adjust
         branch or plex only viewspec%
      % assimilate group to table of contents %
         place _ ptr ;
         last _ getsuc (place) ;
         cea (ptr, getsuc(ptr), getail(ptr), 0, vspec,
```

```
      da,davspc2, da,dausqcod, da,dacacode); %command
      execute assimilate procedure, using modified copy
      of first viewspec word and the rest from the
      display area descriptors%
   % for all statements in table of contents %
      UNTIL (place _ getnxt(place)) = last DO
      dotoc(place) ;  %turns statement into line for
      table of contents%
   % move table of contents to under st 1 %
      cmg (ptr, getsuc(ptr), getprd(last), $"d");
      %command move group procedure%
   % recreate display %
      IF nlmode=fulldisplay THEN alldsp() ELSE crlf() ;
   RETURN ;
   END.
(dotoc) PROCEDURE (stid) ; %passed stid, replaces
statement with table of contents line %
   % declarations %
      LOCAL length;
      LOCAL STRING dots[70], stnum[50], st[2000] ;
      LOCAL TEXT POINTER end ;
   % initializations %
      length _ st.L _ stnum.L _ 0;
      *dots* _
      "..................................................
      ...................." ;
   % get st number %
      stnum.L _ 0;
      fechno (stid, $stnum); %put statement number in
      string%
   % get first line %
      *st* _ SF(stid) SE(stid) ;
      length _ (65 - (3*getlev(stid)+stnum.L)); %maximum
      length%
      IF length < st.L THEN
         BEGIN
         st.L _ length ;  %truncate statement%
         FIND SE(*st*) [NP] "end > ; %back up to end of
         last word%
         *st* _ SF(*st*) end ;
         END;
   % format string %
      dots.L _ (length + 2) - st.L; %calculate number of
      dots%
      *st* _ *st*, *dots*, *stnum*; %constuct table of
      contents string%
   % replace statement %
      ST stid _ *st* ;
```

```
      RETURN;
      END.
    FINISH toc
```

Procedures Used in Examples; references taken from <NLS>SYSGD 5g3e

Format of references:

(proc-name) (link to source code) st-num-of-source-code

(formal, parameters, if, any)

comment taken from source code file

(alldsp) (nls,dspgen,alldsp) 3A
recreate display for all display areas
(cea) (nls,corenl,cea) 7A
(target,src1,src2,levstg,vspec1,vspec2,usqcod,cacode)
Core NLS Assimilate Command
(cis) (nls,corenl,cis) 9H
(stid,astrng,levstg)
Core NLS Insert Statement Command
(cmg) (nls,corenl,cmg) 11L
(stid1,stid2,stid3,levstg)
Core NLs Move Group command
(crlf) (nls,inpfbk,crlf) 6G
type a carriage return-line feed
(dn) (nls,inpfbk,dn) 8E1
(astrng)
display string in name area
(fechno) (nls,seqgen,fechno) 4J
(stid,astr)
Puts statement number of stid in string. Give the STID as the first argument, and the address of the string which is to contain the statement number as the second. The statement number will be built in the string. If the structure is not intact or the statement vector cannot be built, a call to RERROR or an EXCEED CAPICITY ERROR may result.
(getail) (nls,strmnp,getail) 10A
(stid)
Given an stid, this procedure returns the stid of the tail of the current plex
(getlev) (nls,seqgen,getlev) 4I
(stid)
Called with STID, returns level of that statement.
(getnxt) (nls,strmnp,getnxt) 10G
(stid)

This procedure finds the sequentially "next" statement, i.e. the substatement, successor, or successor of up, etc, of the stid passed as argument. Ignores all viewspecs.

(getprd) (nls,strmnp,getprd) 10D
(stid)
Given an stid, this routine returns the predecessor; if the psid heads a plex, the stid itself is returned

(getsuc) (nls,filmnp,getsuc) 2H1
(stid)
The stid for the successor field is returned. If there is no successor, the stid of the up is returned.

(lda) (nls,dactrl,lda) 5J
returns address of display area where bug resided at last command terminator

(tbug) (nls,txcmnd,tbug) 5A
(ptr)
given the address of a text pointer, gets an address selection from the TNLS user and puts it in the text pointer.

(txtlit) (nls,inpfbk,txtlit) 5B
(astrng)
passed the address of a string, appends text from keyboard input buffer to string

(typeas) (nls,inpfbk,typeas) 6C
(astrng)
Given the address of a string, types the string on the user's teletype.

PART FOUR: Advanced L10 Programming 6

Section 1: Executable Programs 6a

Introduction 6a1

For most applications, it is sufficient to accept statements one at a time from the sequence generator and assume an initial character position of the beginning of the statement (a Content Analyzer program). When one has more complex applications, one may have to write more complex programs which are explicitly passed control. These are not called by the sequence generator but are passed control from the Programs subsystem (see Section 9 -- 4i2). Therefore they must provide themselves with statements on which to work. They should not return a value (as did the simpler Content Analyzer type programs), but should just return control to the calling subsystem. All the capabilities described above are available to such programs. In addition, the program may skip around files, between files, and may interact with the user. 6a1a

Moving Around a File 6a2

Generally, at least one simple variable or a text pointer will have to be declared to hold the statement identifier (stid) of the current statement. (The first word of a text pointer is an stid.) Assume the simple variable with the name "stid" has been declared for the purpose of the following discussion. 6a2a

In the NLS file system, two basic pointers are kept with each statement: to the substatement and to the successor. 6a2b

If there is no substatement, the substatement-pointer will point to the statement itself.

The procedure getsub returns the stid of the substatement. To do something to the substatement if there is one:

```
IF (stid := getsub(stid)) # stid THEN something..;
```

stid is given the value of the substatement-pointer, then the old value of stid is compared to the new. If they are the same, then there is no substructure. If they are different, you have the stid of the substatement and can operate on it.

If there is no successor (at the tail of a plex), the successor-pointer will point to the statement UP from the statement (i.e. the statement to which the current statement is a sub).

The procedure getsuc returns the stid of the successor (or Up).

To move to the successor:

```
stid _ getsuc(stid);
```

Given these two basic procedures, a number of other procedures have been written and are part of the NLS system. All of the following procedures take an stid as their only parameter, and do nothing but return a value, usually a stid. If the end of the file is encountered, these procedures return the global value "endfil". 6a2c

```
getup(stid)  - returns the stid of the up
getprd(stid) - returns stid of the predecessor
getnxt(stid) - returns stid of next statement or endfil
getbck(stid) - returns the stid of the back or endfil
gethed(stid) - returns stid of the head of the plex
getail(stid) - returns stid of the tail of the plex
getend(stid) - returns the stid of the end of the
               tail of the plex
getftl(stid) - returns TRUE if stid is tail of plex,
               else FALSE
getlev(stid) - returns level of statement
```

Once you have the stid of a statement, you may operate on it as in Content Analyzer programs. 6a2d

E.g. FIND SF(stid) $NP "ptr...

Input/Output 6a3

Input and output must be handled quite differently for TNLS and DNLS. There are three system globals which may prove of service in making this distinction: 6a3a

```
    fulldisplay
    typewriter
nlmode - the current value, either fulldisplay
         or typewriter
```

Example:

```
IF nlmode=fulldisplay THEN something ELSE otherthing;
```

There are a few procedures that work in both DNLS and TNLS: 6a3b

These return the ASCII value of a character from the keyboard input buffer:

```
input()  - get next character from keyboard
           input buffer
inpcuc() - get character, forced upper-case,
           from the keyboard input buffer
lookc()  - returns the next character in the
           input buffer without advancing the
           buffer pointer (i.e. what the
           next input() will return)
```

```
dismes(type,astring) - given a type number and the
           address of a string, will print the message
           on the user's teletype or (in DNLS) display
           it in the teletype simulation window (above
           the command feedback line).
   type=0: clear message area; astring not necessary
       =1: put out message and leave it there
       =2: display message for a few seconds (same
           as 1 for TNLS)
```

```
            >1000: display for n microseconds (same as
                1 for TNLS)
        Remember, a dollar sign preceding a variable means the
        address of that variable.
           e.g.  dismes (2, $strvar) ;
        A temporary string may be declared in the procedure call
        for the use of that procedure alone:
           dismes (1, $"string of text to be displayed") ;
     levset(stid,astring) - given an stid and the address of
                 a string containing levadj characters (u's
                 and d's), evaluates levadj and returns a
                 target stid and 0 if new statement is to be
                 down from target or 1 if successor. Used
                 in routines which insert statements.
  TNLS                                                            6a3c
     There are no standard L10 constructs for TNLS I/O. The
     following procedures should be of help:
        txtlit(astring) - passed the address of a string,
                    appends text from keyboard to string
        typeas(astring) - passed the address of a string,
                    types string on tty. The programmer
                    may declare a temporary string in cases
                    like this, e.g.
                          typeas ($"this will print out") ;
        crlf()      - type a carriage return-line feed
                    on the tty (You may also have a
                    carriage return in a string passed
                    to typeas.)
        levadj(stid,astring) - given an stid and the address
                    of a string variable, gets a string of
                    levadj characters (u's and d's) from the
                    user and puts them in the string
        tbug(atp)   - passed the address of a text pointer,
                    gets address from user
        tbug2(atp1,atp2) - get two bugs, the second relative
                    to the first
  DNLS                                                            6a3d
     There are some standard L10 statements for DNLS I/O:
        INPUT
           INPUT may be followed by any sequence of the
           following; backup within the command (backspaces) is
           handled automatically:
              BUG ptr  - get a bug selection from the cursor
                       and store the resulting text pointer
                       (ptr must be a text pointer, not an
                       stid) in ptr
              STID ptr - get a bug from the cursor or a SP
                       followed by a statement name, number
```

```
                or SID, and store the resulting text
                pointer in ptr
      LEVADJ str - get a sequence of level adjust
                characters (u or d) and store them
                in the string str
      TEXT str - get a string of characters (up to a
                CA or Center-Dot), echoing them in
                the text area of the display, and
                store them in the string str
      STRING str - like TEXT except echoes in the
                name area
      NAME str - get a string of characters forced
                upper-case, echoing them in the name
                area of the display, and store them
                in the string str; the characters may
                be typed in or a word may be bugged
      WORD str - like NAME except not forced
                upper-case
      NUMBER str - like NAME except inputs a number,
                typed or bugged
      statement; - any standard L10 statement,
                followed by a semicolon if necessary
                to delimit the end of the statement;
                the statement will be executed at
                that point in the input sequence
      char - succeeds if specified character is
                input; may be any of the characters
                mentioned under "Primitives" or
         CA  - Command Accept
         CD  - Command Delete
         ALT - Alt Mode, Escape
         BC  - Backspace Character
         BW  - Backspace Word
         C.  - Center Dot
   Example (a simulation of a subset of the Replace Text
   command):
      INPUT BUG b1 BUG b2 (BUG b3 BUG b4 CA flag_TRUE; /
      TEXT lit CA flag_FALSE) ;
      IF flag THEN ST b1 b2 _ b3 b4
              ELSE ST b1 b2 _ *lit* ;
DSP -- the Command Feedback line
   One may control the text of the command feedback line
   with the following L10 statement:
      DSP ( dsp-element ) ;
      where dsp-element is any sequence of the following:
         < - clear command feedback line
         _ - move arrow to far left
         " - set arrow under start of nxt word
```

&SRI-ARC 31-OCT-74 14:54 24258
ARC Rev. 1 NOV 74

..., - replace last word currently in command feedback line with next word
a word - including letters or digits only; will be added to command feedback line
To display special characters, surround them with quotation marks.
Additionally, the following procedures may be of service; some take no parameters:
an() - turn arrow on
af() - turn arrow off
qm() - turn question mark on
qmoff() - turn question mark off
arm() - arm the bug cursor
disarm() - disarm the cursor
dn(astring) - given the address of a string, will display the string in the name register; as with dismes(astring), you may declare a temporary string as the argument
litdpy(astring) - given the address of a string, will clear file display area and display contents of the string
rstlit() - restores file area after a litdpy()

Calling NLS Commands 6a4

A program may execute any of the standard NLS commands by calling the same procedure that the command parser calls for each command. These procedures are called the "core" procedures. They are listed in <NLS>SYSGD. Their names begin with the letter "c", followed by the initials of the command, e.g. Insert Statement could be executed by calling the procedure "cis". 6a4a

Usually the required arguments can be discovered by knowing the command and by looking at SYSGD. For example, the formal parameters to the procedure "cis" are (stid,astrng,levstg). Obviously, the procedure wants a target stid, the address of a string of text to be inserted, and the address of a string holding level adjust characters (u's and d's). 6a4b

Much can be learned by looking at the code of the core procedure. You can see what procedures it in turn calls to discover how the command is actually performed. But most importantly, you can find out what the procedure returns. The RETURN statement for "cis" look like: 6a4c

```
RETURN(stid);
```

from which it can be inferred that the procedure returns the stid of the newly created statement. 6a4d

When you are not sure what the arguments mean, a good way to find out is to see where the command parser picks up the information. You can follow through the parsing of a command by begiining with <NLS>NCTRL, the actual command parsing procedure. 6a4e

Tracing a command from <NLS>NCTRL is also valuable in finding out how the system performs an operation which you would like your program to do. For example, if you wish to parse a link and open the given file, you might learn how to do it by following the Jump to Link command through. 6a4f

Opening Files 6a5

When you ask the user for an address or bug, you don't have to open the file; you have a handle on it with the stid the user gives you. There may be times, however, when you wish your program to open a file. There is a procedure which does this: 6a5a

```
open (jfn, astring);
```

You should pass zero as the jfn, and the address of a string containing the name of the file to astring. This procedure will return the file number. If the file is not already open, it will open it. It will also fill out the string with the complete file name if you do not specify the directory or version number. The usual sequence of steps to open a file is as follows: 6a5b

```
% stid has been declared as a simple variable or text
pointer%
stid _ orgstid; %orgstid is a global with all zeros except
in the stpsid field, where it has the stpsid of the origin
statement (the same for every file)%
*str* _ "<dirname>filename.nls"; %str is of course a
declared string variable%
stid.stfile _ open (0,sstr);
```

At the end of your program, you should close any files that you have opened. Use the procedure: 6a5c

```
close (filnum);
e.g.
        close (stid.stfile);
```

Another common operation is to access the statement (file) in which the CM (or bug) was at the time of the last Command Accept (or other command terminator). This is stored in the system, and can be accessed with the following procedure call: 6a5d

```
stid _ lccsp() ;
%then, if you wish to set the stpsid to the origin of that
file, you could say%
   stid.stpsid _ origin ; %origin is a global with the
   stpsid of the origin statement in it%
```

Other Useful Procedures 6a6

alldsp() - DNLS only; recreates display in all display areas, so that user will see changes the program made. 6a6a

A common way to end a program (just before returning) is with a statement like:

IF nlmode=fulldisplay THEN alldsp() ELSE crlf() ;
answer() - waits for a yes or no from the user; returns 0 if no, and 1 if yes. 6a6b
astruc(astring) - given the address of a string, sets the string to upper case. 6a6c
fechno(stid,astring) - given an stid, appends the statement number string to the string variable whose address is passed. 6a6d
fechsig(stid,astring) - given an stid, appends the statement signature to the string variable whose address is passed. 6a6e
getdat(astring) - given the address of a string, appends date and time to string. 6a6f
grptst(stid1,stid2) - checks that two stid's specify a legal group; returns them ordered or else an "illegal group" signal is generated. 6a6g
lookup(ptr,string,type) -given the address of a text pointer, the address of a string, and a type, will do a variety of searches (in the process destroys string and changes pointer): 6a6h
type = nametyp: non-sequential search for statement of name given in string; returns stid and sets pointer to stid or else returns endfil in both places
type = nxtname: like name, also a non-sequential search, but starts from place in file ring to which ptr points
type = seqname: starting with the statement following the one refered to by the ptr, does a sequential search of the file for the given name; returns stid or endfil in pointer
type = contnt: does a sequential search of the file, beginning with the character following the pointer, for a statement with the content of the string; returns stid or endfil in pointer
type = contls: same as contnt, but looks only in statement holding pointer
type = wordtyp: same as contnt, but looks for word given in string
type = sid: pass an SID instead of the address of the string; searches for statement with that SID and returns in pointer and as procedure value the stid or endfil

Globals of Interest: 6a7
initsr is the login ident of the person currently using the program. 6a7a
inpstp is incremented every time the user types a <control-s>; this can be used as a program interrupt mechanism; i.e. you can set it to 0 at the beginning of the program and then check it at the start of each loop of your program. 6a7b

Section 2: Error Handling -- SIGNALs 6b

Introduction 6b1

When an NLS system procedure fails to perform properly, it may generate an error signal. Every signal has a value. When a signal is generated, control is passed back to the last signal trap in effect. If no explicit program control statement (e.g. RETURN, GOTO) is given in that signal trap, a new signal will be generated. If the error is not dealt with, the signal will eventually bubble all the way back to the core NLS system and the program will stop. You may trap signals and regain control by setting up the response in advance. 6b1a

Trapping Signals 6b2

To trap error signals of any error value: 6b2a

```
ON SIGNAL ELSE statement ;
e.g.  ON SIGNAL ELSE
         BEGIN
         dismes(2,$string);
         RETURN;
         END;
```

It is a good idea to set up a signal response before calling any NLS system procedures. Once the signal response is set, it remains in effect and will be executed whenever a signal is received through the end of the procedure or until it is changed. Any subsequent ON SIGNAL statements will at that point change the signal response. 6b2b

A signal trap set inside a loop will only remain in effect within the loop.

Only signals generated by procedures below (e.g. called by) your procedure will be trapped by your procedure's signal trap. It will not trap signals generated in the same procedure. 6b2c

The signal response may be any (block of) L10 statement(s). It will be executed, then 6b2d

- if you have an explicit program control statement (RETURN, GOTO, EXIT LOOP), control will be passed accordingly, or
- if the signal trap includes no explicit program control statement, another signal will be generated, and control will pass upward through the stack of procedures called until it encounters another signal trap.

Thus, if you wish to resume control in the current procedure, the signal trap will have to end with a GOTO statement pointing to an appropriately labeled statement. This is one of the few places where a GOTO is really necessary. 6b2e

If the signal trap applies to a loop, an EXIT LOOP or REPEAT LOOP is a valid signal program control statement. 6b2f

Cancelling Signal Traps 6b3

If, after setting up a signal response, you wish to cancel it so that the signal will just bubble on up, you may do so with the statement: 6b3a

```
ON SIGNAL ELSE ;
```

It may be subsequently reset by another ON SIGNAL statement. 6b3b

Specific Signals 6b4

When a signal is generated, an NLS system global variable, sysgnl, is given a specific value (the value of the signal). Each value represents a certain type of error. Also, a system global variable, sysmsg, is given the address of a string which holds an error message. 6b4a

The above constructions react to any signal, no matter what its value may be. The ON SIGNAL statement can be used much like a CASE statement if you wish to trap specific signals: 6b4b

```
ON SIGNAL
   =constant: statement;
   =constant: statement;
      ...
   ELSE statement;
e.g. ON SIGNAL
         =ofilerr: %open file error%
            BEGIN
            IF sysmsg THEN dismes(2,sysmsg);
            RETURN;
            END;
         ELSE %any other error signal%
            BEGIN
            dismes(2,$"Error");
            RETURN;
            END;
```

The current signal constants can be found in (nls,const,). The common reason for using this specific signal treatment is when you call a procedure which you know will generate a certain signal value under certain conditions. In such a case, you can learn the signal constant of concern from the SIGNAL statement which generates it. 6b4c

Generating Signals 6b5

You may generate a SIGNAL in a procedure by the statement: 6b5a

```
SIGNAL (value, astring) ;
```

where value is the value of the signal (perhaps a system global) and astring is the address of a string holding the error message. If the second parameter is omitted, it will be assumed to be zero and no message will be printed. The first parameter is mandatory; every signal must have a value. 6b5b

Examples:

```
SIGNAL (ofilerr, $"Couldn't open your file.") ;
SIGNAL (2) ;
```

Another way to generate a SIGNAL is by calling the procedure "err". It takes one parameter, a number representing the typeof error. It will generate a SIGNAL of the value "errsig" (a system global) and will set up a message depending on the error number you pass it. The standard error messages are: 6b5c

```
errno = 1: "File copy fails";
      = 2: "Open scratch fails";
      = 3: "Cannot load program";
      = 4: "I/O Error";
      = 5: "Exceed capacity";
      = 6: "Bad file block";
      = 7: "Not implemented";
```

If you pass it the address of a string as the error number, it will signal using that address for sysmsg, and that string will be printed.

Be careful not to call err and then trap its SIGNAL in that same procedure. You might say: 6b5d

```
ON SIGNAL
   =errsig: NULL;
   ELSE ...
```

ASCII 7-BIT CHARACTER CODES

Char	ASCII	Char	ASCII	Char	ASCII	Char	ASCII	Char	ASCII
Tab	011	/	057	B	102	U	125	h	150
LF	012	0	060	C	103	V	126	i	151
FormFeed	014	1	061	D	104	W	127	j	152
CR	015	2	062	E	105	X	130	k	153
SP	040	3	063	F	106	Y	131	l	154
!	041	4	064	G	107	Z	132	m	155
"	042	5	065	H	110	[	133	n	156
#	043	6	066	I	111	\	134	o	157
$	044	7	067	J	112	]	135	p	160
%	045	8	070	K	113	^	136	q	161
&	046	9	071	L	114	_	137	r	162
'	047	:	072	M	115			s	163
(	050	;	073	N	116	a	141	t	164
)	051	<	074	O	117	b	142	u	165
*	052	=	075	P	120	c	143	v	166
+	053	>	076	Q	121	d	144	w	167
,	054	?	077	R	122	e	145	x	170
-	055	@	100	S	123	f	146	y	171
.	056	A	101	T	124	g	147	z	172

L10 Users' Guide

Augmentation Research Center

1 NOV 74

Stanford Research Institute
333 Ravenswood Avenue
Menlo Park, California 94025

For the most recent online version of the L10-Guide, see
<USERGUIDES,L10-Guide,>

(J24258) 31-OCT-74 14:54;;;; Title: Author(s): Augmentation Research Center /&SRI-ARC; Distribution: /DIRT([INFO-ONLY]) ; Sub-Collections: DIRT SRI-ARC NIC; Clerk: POOH; Origin: < USERGUIDES, L10-GUIDE.NLS;319, >, 29-OCT-74 11:41 NDM ;;;;

####;

Inventory of offline documentation found in room J2028

Offline copies of the following Documentation can be found on the shelves in room J2028. Please help yourselves. If you take a last copy, please let me know. A more complete list of all ARC Documentation will follow shortly.... 1

ARC Tenex Users' Guide 1a

TNLS-8 Primer 1b

Dex Primer (to be updated) 1c

Dex User Guide (to be updated) 1d

L-10 Users' Guide (to be updated) 1e

L-10 Documentation (to be updated) 1f

Output Processor Users' Guide (to be updated) 1g

NLS-8 Command Summary 1h

NLS-8 Equivalents of NLS-7 Commands 1i

CML Documentation 1j

NLS File Structure 1k

NDDT Symbolic Debugger User's Guide (to be updated, commands are in old syntax pending revisions of commands) 1l

Proposed NLS Code Format and Documentation Standards 1m

Links in xnls 1n

Coordinated Information Service for a Discipline-or Mission-Oriented Community 1o

The Augmented Knowledge Workshop 1p

Advanced Intellect-Augmentation Techniques 1q

TNLS-8 Quick Reference (Cue Card) 1r

Inventory of offline documentation found in room J2028

(J24259) 21-OCT-74 14:43;;;; Title: Author(s): Anne Weinberg/POOH; Distribution: /JOAN([ACTION] please put a copy of this in the DIRT notebook) SRI-ARC([INFO-ONLY]) RSR([INFO-ONLY]) MAP2([INFO-ONLY]) ; Sub-Collections: SRI-ARC; Clerk: POOH; Origin: < WEINBERG, HARDDOC.NLS;1, >, 21-OCT-74 14:22 POOH ;;;;####;

Hostnames, Liaison lists, and RFC distribution list

The NIC (Adrian and I) maintains the official hostnames list and the Liaison list. Gail Hedtler and Jerry Burchfiel are maintaining a distribution list suitable for RFC distribution a subset of which is the Liaison list. The NIC lists are: <NETINFO>LIAISON.TXT, <NETINFO>LIAISON-SNDMSGS.TXT and <NETINFO>HOSTS.TXT all at Office-1 and available for ftp through nicguest password ARPA. Please direct all questions about hostnames and liaison lists to me and not to Alex McKenzie as he is getting annoyed at dealing with several people. Contact Jon Postel (as group co-ordinator) to get your name on the RFC distribution list. Feel free to use my lists anytime - I try to update them weekly if new information is received. Thanks, Jake 1

Hostnames, Liaison lists, and RFC distribution list

(J24260) 21-OCT-74 16:35;;;; Title: Author(s): Elizabeth J. (Jake) Feinler/JAKE; Distribution: /SRI-ARC([INFO-ONLY]) ; Sub-Collections: SRI-ARC; Clerk: JAKE;

New XNLS , Monday Oct 21 1974

NLS BUGS FIXED Monday OCT 21, 1974 1

I brought up an Xnls on Monday night with the following modifications: 1a

TNLS: 1a1

the following commands have been removed 1a1a

Jump [to] File <SPACE> 1a1a1

Jump [to] File <OK> 1a1a2

Jump [to] Name <OK> 1a1a3

The subsitute Command no longer gives a double prompt when one replies "y" to the question, <Finished?>. 1a1b

DNLS & TNLS: 1a2

Replying "N" to the question <Insert Number List?> in the Reserve Number Command of the Sendmail subsystem no longer produces a question mark 1a2a

The changed source files are psupport and syntax. I changed both of these files in both directories NLS and NIC-NLS. 1b

New XNLS , Monday Oct 21 1974

(J24261) 21-OCT-74 17:58;;;; Title: Author(s): David S, Maynard/DSM; Distribution: /JMB([ACTION]) KIRK([ACTION]) JDH([ACTION]) DSM([INFO-ONLY]) EKM([INFO-ONLY]) ; Sub-Collections: SRI-ARC; Clerk: DSM;

DVN 21-OCT-74 19:39 24263

Alba Amicorum

Could yo do me the favor of asking Caroline what "alba amicorum" might mean in the context of Christian religious books? 1

Alba Amicorum

(J24263) 21-OCT-74 19:39;;;; Title: Author(s): Dirk H. Van Nouhuys/DVN; Distribution: /KIRK([ACTION]) ; Sub-Collections: SRI-ARC; Clerk: DVN;

Liaison at BRL

Hi, Stan,

Wonder if you could tell me whether Mike Romanelli is still the Technical Liaison at BRL. Someone said they thought he was no longer there and it is a little embarrassing to send a message saying 'are you still there?'. Would appreciate any input you might have. Thanks, Jake 1

Liaison at BRL

(J24264) 22-OCT-74 11:21;;;; Title: Author(s): Elizabeth J. (Jake) Feinler/JAKE; Distribution: /DFT([ACTION]) ; Sub-Collections: SRI-ARC; Clerk: JAKE;

Feedback

Doug, Is it all right if I proceed with the Arpanews idea (Hjournal, 24049, 1f:w) I outlined to you earlier. I need to know before Weds., Oct. 23, because Kjell Samuelson may be coming through on Thursday and I would like to let him know what is happening. Also, I need to contact the others too. Thanks, Jake. 1

Feedback

(J24265) 22-OCT-74 11:45;1, >, 23-OCT-74 09:37 XXX ;;;; Title:
Author(s): Elizabeth J. (Jake) Feinler/JAKE; Distribution: /DCE([
ACTION]) ; Sub-Collections: SRI-ARC; Clerk: JAKE; Origin: <
FEINLER, DOUG.NLS;2, >, 22-OCT-74 11:42 JAKE ;;;;####;