-*-Text-*- Documentation file for DDT

^V
File: DDT Node: Top Next: Conventions Up: (DIR)

DDT Primer and Reference Manual
describing DDT version 650.

DDT is the top-level command interpreter most often used on the Incompatible Timesharing System (ITS). It provides three main types of service to the user: invocation of system programs, manipulation of files, and aid in debugging. ITS itself has no command processor, and can be used only via a program. What appears to a user as the "monitor command level" of ITS is actually DDT.

This file (the primer, INFO;DDT >) treats the general aspects of DDT, grouping commands according to function; then, in the "reference section" (.INFO.;DDT ORDER), all the commands will be described in detail, in alphabetical order. Before using a command in a complicated way, refer to the detailed description to verify that it will function as intended.

* Menu:

^V

File: DDT, Node: Conventions, Previous: Top, Up: Top, Next: Rubout

Conventions Used in This Document

When examples of DDT commands are given, upper case letters are intended to be typed in exactly as shown. Angle-brackets ("<" and ">") enclose "meta-variables" which stand for arguments to be supplied by the user. Uparrow ("^"), known to some as circumflex, is used for representing control characters: "^A" signifies a control-A. Other non-alphanumeric characters are to be typed in as shown; some special characters that would be confusing if represented by themselves are represented by "meta-constants", such as "<cr>" for a carriage-return. An alternate representation for a <cr> would be "^M". Other characters sometimes represented by meta-constants include <backspace>, <tab>, <lf>, <rubout>, <space>, and <comma>. Altmode is represented by itself, and its appearance is ◊. These conventions may be violated in some places, but there will be a note to that effect near the scene of the crime.

Numbers are octal, unless followed by a decimal point ("."), in which case they are decimal (this is the same convention as DDT uses for input). Following the ITS 1.5 manual, "bit <m>.<n>" means the <n>'th bit from the bottom of the <m>'th 9-bit byte from the bottom, out of the four 9-bit bytes in the 36.-bit word. Thus, bit 1.1 is 1, bit 1.3 is 4, bit 1.7 is 100, and bit 2.2 is 2000. Bit 3.1 is 1,, and bit 4.9 is the sign bit, 400000,, . The last sentence illustrates another convention, that <lh>,, is <lh> shifted left 18. bits, or put in the left halfword.

The term "ref <command>" will mean "see under <command> in the appropriate part of the reference section". "See the reference section <part>" will mean to refer to the named part of the reference section.
v

File: DDT, Node: Rubout, Previous: Conventions, Up: Top, Next: Startup

DDT Type-in Conventions

DDT has several special action characters that have a specific effect on whatever command is being typed, instead of simply becoming part of the command. They are used for editing input, for aborting operations, or for controlling where DDT's output goes.

The most important input editing character is <rubout> (ASCII code 177), which deletes the last input character. If possible, it will be erased from the screen; otherwise, it will be typed back out to show what is happening (sometimes <rubout> deletes several characters. When that happens, it types them all out). Other input editing characters are ^D, which cancels ALL unprocessed type-in (in other words, as much as it's possible to cancel), and ^L, which clears the screen and then retypes the unprocessed input.

^S turns off DDT output to the terminal for the command in progress, and other all commands before the ^S itself. ^S is the right character to use if you don't want to see all of a file or directory listing you have printed. Many DDT commands will stop executing if they are silenced, if they think that they are no longer doing anything useful.

^G is a powerful abort command. It will return DDT instantly to its main command-reading level at almost any time. It is so strong that it is dangerous, because it can catch DDT in the middle of updating a data base, such as a job's symbol table or the list of jobs. ^G should be used as a last resort, when DDT hangs up in the middle of a command and ignores other input. In less extreme situations, ^S is usually enough.

One time when ^G is ineffective is when DDT has given control of the terminal to another program. To stop the other program and tell DDT to resume reading commands, the character ^Z may be typed on the terminal (CALL, on TV terminals). ^Z or CALL is interpreted by ITS itself, and causes a special unconditionally fatal interrupt to the job that has control of the terminal. DDT notices the fatal interrupt and takes control; it never sees the ^Z per se. There is no reason to type ^Z when DDT itself has control of the terminal (assuming the DDT is top level, as usual), and it causes an error message "??" to be printed.

The characters ^W and ^V are used to turn terminal output off and on for long periods of time -- for example, if it is desired to execute several commands without any printout on the terminal. ^W turns typeout off, and ^V turns it back on. Typeout will also be turned on if any DDT command reports an error, or if any other program run by DDT returns abnormally. There are also characters ^E and ^B that turn off and on output to a script file. See the reference section for more details.

^_ (BACK-NEXT, on TV's) is another character that is interpreted by ITS directly. It is used for communicating with other users, and for specifying various options connected with ITS's handling of the terminal. See the file .INFO.;ITS TTY for details.
v

File: DDT, Node: Startup, Previous: Rubout, Up:Top, Next: Login

When DDT Starts Up

This section's title is purposefully ambiguous.

Since ITS has no internal command processor, it is impossible to do anything with a terminal unless some job is deliberately taking input from it. A free terminal is not being read by any job and what is typed on it is ignored. There is an exception, of course, or else it would be impossible to log in. When the character ^Z is typed on a free terminal (or CALL, on a TV) ITS itself loads a copy of DDT to read commands from the terminal. Through DDT, all of the system's facilities are accessible.

On ITS, every running copy of a program must reside in a job, which is a virtual machine much like a PDP-10. Every job has two names: the UNAME which identifies the user it belongs to, and the JNAME which distinguishes between the jobs belonging to one user. The first job any user has is the one that ITS puts his DDT in; that job's JNAME is always "HACTRN" (a parody of "FORTRN"). Other jobs may be created by DDT to run other programs in. Note that "HACTRN" is the name of the job; "DDT" is the name of the program that normally runs in it. It is easy to make another copy of DDT run in a different job, and not very hard to get some other program running in the job HACTRN.

The first thing DDT does when it starts up is print some status information about the system, including the name of the machine being used (AI, ML, DM, or MC), the version numbers of the DDT and ITS that are running, and the number of users on the system. This information can be obtained at any later time with the :SSTATUS and :VERSION commands. In addition, DDT prints the files SYS:SYSTEM MAIL and (except for network users) SYS:LOCAL MAIL, which contain notices so urgent that everyone should see them before using the system.

The "fair share" included in the startup statistics is an indication of what fraction of the total CPU time is available to any one user. It is a measure of the load level on the system. If it is down to 10% or so, you might prefer to wait for the load to be less.

File: DDT, Node: Login, Previous: Startup, Up: Top, Next: Files

Logging in and out

Logging in is the way a user identifies himself to the system and the other users. One should generally log in soon after the beginning of any session. Although being logged in is not actually necessary for most things, it is both convenient (mail you send will contain your name, etc.) and respectful to the other users.

Logging in involves specifying a UNAME, or user name, of 6 characters or less (preferably letters). Each user has his own customary UNAME (or several). If you don't have your own UNAME, pick one that nobody else uses (do :WHOIS <uname><cr> to find out whether a UNAME is in use already). Most users use their initials or one of their names for a UNAME. For more information on what DDT does with the UNAME specified, see the reference section "DDT's UNAMEs".

There are two commands for logging in: :LOGIN, which is a "colon command", and ◊U, which is a "DDT command". Colon commands all start with a colon, which is followed by the name of the command. After the command name come the arguments, if any. In the case of :LOGIN, the argument is the UNAME to log in as. After the arguments, a <cr> ends the command. If DDT is given a colon command with an unrecognized name, it tries to run the system program (if any) with that name. "DDT commands" all contain either a special character or at least one altmode (or both); they can take either prefix or suffix arguments depending on the command. As you see, the term "DDT command" is ambiguous, and can mean any command that DDT understands, or can exclude colon commands.

The dichotomy of colon commands and DDT commands reflects DDT's evolution. Although most often used nowadays for file manipulation and for running system programs, DDT's original use was in debugging. The other functions were added when ITS was written and DDT was chosen as its executive program. When DDT was used only for debugging, there were no colon commands. Nowadays, the more cryptic DDT commands are (usually) assigned to operations that either pertain to debugging or are frequently used, while the longer but mnemonic colon commands are assigned to operations that are either obscure and infrequent or needed by unsophisticated users. Many important operations have both a DDT command to save typing for expert users and a colon command for new users. In such cases, the DDT command is called the "short form". Logging in is such an operation, with the colon command ":LOGIN" and the short form "◊U".

ITS normally knows the characteristics of hardwired terminals. For non-local terminals, it may not know them automatically. In that case, before you log in, you should tell ITS the terminal characteristics using the TCTYP program. In the simplest case, that is done by :TCTYP <terminal type><cr>.
For example, :TCTYP EXECUPORT<cr> tells ITS to treat the terminal as an execuport. :TCTYP<cr> by itself tells what ITS belives about the terminal at the moment. :TCTYP HELP<cr> will print more information on the TCTYP program.

When you log in, if you have received mail from other users, or if there are announcements on the system that you have not yet seen (see :MSGS), DDT will normally offer to show them to you. The offer will look like "--Mail--" or "--MSGS--", and the proper responses are <space> meaning "yes, show me the mail or the MSGS", or anything else, meaning "save them for later". This scheme has the property that an unexpected offer never causes interference: if you type a command after logging in, without waiting to see whether DDT offers mail or not, DDT will obey the command and skip the printing of the mail. DDT makes other offers at various times. They all begin and end with two dashes, and all are answered approximately the same way (<space> for "yes", anything else for "no"). See the reference section

"Unsolicited Offers" for full details. Meanwhile, :PRMAIL will print mail at any time, and :MSGS will print new system messages.

When the :LOGIN or ◊U command is finished, it prints a "*". Many commands do this, to indicate that they are finished. The "*" is known as the "prompt character", but, unlike the prompt characters of many other programs, it means "operation completed" rather than "ready for more input". Input may be typed at any time, and will be saved by ITS until it is wanted. Some people like to change the "*" to some other string. Ref ":DDTSYM PROMPT" for how.

DDT has an "init file" feature that allows a user to customize his DDT. The init file is a file of DDT commands that will be executed automatically on logging in. For details, see the section DDT Commands from Files.

Here is a description of the :LOGIN command, in the format that will be used throughout this document:

:LOGIN <name> or <name>◊U

logs in as <name>. Then, if there is a DDT init file, it is executed as DDT commands. Otherwise, DDT will offer to print user <name>'s mail and any system messages he hasn't yet seen. After logging in, a second :LOGIN is not allowed. If <name> is already logged in somewhere else, DDT will automatically try to log in as <name>0, <name>1, etc. instead, until it finds a name that isn't in use. But even if it logs in as <name>0, DDT will still remember that it was "supposed" to log in as <name> (see the XUNAME in the reference section "DDT's UNAMEs"), so it will print <name>'s mail instead of trying to find mail sent to <name>0.

:CHUNAM <uname>

changes DDT's UNAME to <uname>. It has much the same effect as logging out and logging back in again as <uname>, except that any user options set in DDT by the previous user remain in effect. The same holds for any ITS options pertaining to the terminal. However, DDT will offer to --Reset All--, and if given a <space> it will make a special effort to reset all such options to their normal states. This involves loading a new copy of DDT from the disk so that all variables in DDT are reinitialized (see ◊U. in the reference section).

When finished using ITS for a time, you should "log out" to free your resources for other users. The operation of logging out is the inverse of typing ^Z and logging in. It destroys DDT and any other jobs it has created (except for any that have been disowned), so be sure to write out any files you are editing before logging out! The terminal becomes free again, and will ignore everything typed on it except for ^Z (or CALL) and ^_. ITS types a message on the console stating that it is free and also giving the time of day.

:LOGOUT or ◊◊U

logs the user out. Only the top-level job (HACTRN) can log out; if the command is given to a DDT in a job that is not top-level, it is an error.

DDT allows the user to have an "exit file" of DDT commands to be executed when he logs out. See :LOGOUT and :OUTTEST in the reference section for full details. One commonly used exit file prints an amusing message or fortune; the file is called STAN.K;ML EXIT and you can use it by making your exit file be a link to it (see :LINK, below).

:DETACH

detaches the user's whole job-tree. The console becomes free just as if it had logged out, but the jobs are not destroyed. They remain in

the system without a console. A detached job is just like a disowned job (see :DISOWN), but got that way differently. The opposite of detaching is attaching. There is a :ATTACH command which performs that operation, but it is too primitive to be convenient in the usual case (don't use it without reading the reference section). However, after logging in DDT automatically checks for the existence of a detached tree and offers to attach to it. After a <space> is typed, the formerly detached tree will be connected to the console (which need not be the same one it was detached from). The new DDT that did the attaching will no longer exist.

If something "goes wrong" with the console, a tree may be detached automatically by the system. For example, if a user coming over the ARPA network closes his connection, his tree will be detached. The same thing happens to all users of TV terminals if the TV front-end PDP-11 crashes. When this happens, the detached tree will be destroyed by the system after an hour goes by, unless it is attached first. As described above, logging back in will automatically tell the new DDT to look for the old detached one and offer to attach it.

There is a program called REATTACH designed specifically for detaching jobs from consoles and attaching jobs to consoles. It can be used to move your jobs to another console, from either the old console, the new console, or someone else's console.
:REATTACH HELP<cr> will print its documentation.
v

File: DDT, Node: Files, Previous: Login, Up: Top, Next: Programs

DDT's File-Manipulation Commands

DDT provides commands for renaming, deleting, and copying files, as well as many other file operations. File commands are followed by the names of the files they operate on. If a command has several arguments, they should be separated by commas. The whole command must be ended with a <cr>. If not enough arguments are given before the <cr>, DDT will read another line of input, after describing what it wants for the next argument. Examples of file commands are

```
:DELETE FOO BAR<cr>
```

which deletes the file named FOO BAR in the current default directory, and

```
:PRINT .INFO.;DDT DOC<cr>
```

which prints the file DDT DOC in the directory .INFO. (this file!).

An ITS file's name has four components: the device name, the SNAME (pronounced "S-name"), and two filenames, called the FN1 and the FN2 (The term "filename" is ambiguous, and can refer either to an FN1 or FN2 or to the whole set of four components). This document is a file, and its FN1 is "DDT", its FN2 is "DOC", its SNAME is ".INFO.", and its device name is "DSK". Commonly used device names include DSK which specifies the machine's disk file structure, and AI, ML, MC and DM, each specifying the disk file structure of the named machine (accessed via the ARPA network if necessary). The meaning of the SNAME depends on the device. For devices DSK and AI, ML, MC and DM, the SNAME is the name of the disk directory that the file is stored in. The FN1 and FN2 identify the file in the selected directory. More generally, the device name and SNAME are called the "directory". A directory listing on ITS lists all the FN1-FN2 combinations that happen to exist, at the moment, under a specific device-SNAME pair.

A "filespec" is the character string that specifies a file's name. In DDT, a filespec can specify any or all of the four components of filenames. A device name should be followed by a colon; an SNAME, by a semicolon. The FN1 and FN2 have no special delimiter, but are identified by their order of appearance. Thus,

```
DSK:RMS;FOO 100
```

specifies DSK as the device name, RMS as the SNAME, FOO as the FN1 and 100 as the FN2.

On ITS, the FN1 of a file usually identifies the program or information it pertains to, and the FN2 is the type of file or the version number of the file. Thus, DDT's source file is SYSENG;DDT 626. Its binary file is SYSBIN;DDT BIN. When a listing is made, it has the file name DDT @XGP. The names BIN and @XGP indicate the type of file (binary and @-listing, respectively), while the common FN1 of the files indicates that they are all logically associated. The FN2 of the source file, namely 626, is composed of just digits; that identifies it as a version number. By convention, when DDT is edited the new version is written out with an FN2 that is 1 larger. To make this convenient, ITS interprets an FN2 of ">" specially: when reading, it refers to the largest version number that exists; when writing, it creates a new file with a version number 1 greater than the largest existing one. The name "<" is also special. It can be used to delete the oldest (actually, lowest-numbered) version of a file.

A filename component omitted in a filespec will be given a default value by DDT, usually the last value specified for that component in either the same command or an earlier one (filenames are "sticky").

For example, after specifying a particular SNAME, all file operations will use that SNAME until a new one is specified. To refer to the last file operated on, a null filespec will serve.

New users are often afraid to rely on filename defaults because they aren't sure how exactly the defaults work. Such fear is reasonable, but DDT has a feature to help dispel it. If altmode is typed when DDT is reading a filename, DDT will print the defaults it is using, and go back to reading. If the altmode FOLLOWS a filespec, DDT will print the full name of the specified file, as obtained by merging the filespec with the defaults. Then it will read another filespec using the printed file names as the defaults. If the printed names themselves are satisfactory, a null filespec is enough. This altmode feature makes it easy to learn just what DDT will do with any filespec, and what defaults it uses.

More information on features available in filespecs may be found in the reference section "Reading of Filenames". More information on filename defaulting is in that section and in the section "Defaulting of Filenames".

Here are the simpler DDT file commands. Notice that some operations have colon commands, some have DDT commands, and some (such as the first one, :PRINT) have both. When there are both, they are equivalent unless otherwise noted. Since all commands that take a filespec as an argument must be terminated by a <cr>, the <cr> is not explicitly mentioned.

:PRINT <file> or ^R <file>

types the file on the terminal. On display terminals, at the bottom of the screen DDT will pause and offer to print the rest of the file: "--More--". A <space> will tell DDT to print the next screenful of the file. Any other command will be obeyed instead of printing the rest of the file.

:DELETE <file> or ^O <file>

deletes the specified file. It cannot be undone. If you screw yourself, there is some chance that the file can be recovered from the magnetic tape backup storage. It is wise to follow <file> with an altmode, so that you see exactly what file will be deleted, and have a chance for a second thought. If DDT thinks you are a novice, it may supply the altmode for you (ref ..DELWARN).

:LISTF <directory> or <dev>^F or <sname>^F

lists the files in the directory. A directory is specified in :LISTF just like a file, except that only the device name and SNAME are meaningful; the FN1 and FN2 are meaningless and if one is mentioned it will be taken to be the SNAME instead. The short form ^F has an idiosyncratic syntax: either a device name or an SNAME may be given as a prefix argument. With no argument, ^F is a very convenient way to list the directory you have been using recently. See the reference section for more details. Note that the text of the directory listing is generated by ITS, not by DDT. Many programs on ITS have the ability to list a directory, since it is so easy to do.

◊◊^F

prints an abbreviated directory listing, showing only the files whose FN1's match the current default FN1. Thus, after :DELETE FOO BAR, ◊◊^F would show all the remaining files named FOO <anything> in the current directory. The abbreviated listing is obtained from the DIR: device, which can make many other types of abbreviated or sorted listings. See the reference section for how to tell ◊◊^F which type to use.

<fn1>◊◊^F

sets the default FN1 to <fn1>, and then does a ◊◊^F to list all the files whose FN1's match <fn1>.

:RENAME <file>,<newname> or ◊◊^O <file>,<newname>

changes the specified file's name. Only the FN1 and FN2 can be renamed, so the second argument to :RENAME must not contain a device name or an SNAME.

:TPL <file>

queues the file for printing on the line printer. Ref also :TPLN.

:COPY <oldfile>,<newfile> or ◊^R <oldfile>,<newfile>

makes a new file containing the same data as an old one. The new file's creation date is made the same as the old one. When you are typing in <newfile>, the defaults are set to <oldfile>, but to its ACTUAL name, rather than to the name you typed. Thus, if you gave FOO > for <oldfile>, then the default for <newfile> might be FOO 4. Thus, it is easy to copy a file keeping the same version number. After the :COPY, the filename defaults revert to <oldfile>, as it was specified, in case you want to delete it or copy it to other places. See also th program INSTALL, which is useful for copying a file from one machine to one or more other machines. Also ref the :COPYN command.

:LINK <newlink>,<linked to>

creates a "link", which is a type of file that actually contains the name of another file to use instead. If RMS;MIDAS MID is the name of a link that points at MIDAS;MIDAS >, then any program that tries to read RMS;MIDAS MID will actually read MIDAS;MIDAS >. Such a link could be created by :LINK RMS;MIDAS MID,MIDAS;MIDAS >. Deleting or overwriting RMS;MIDAS MID will remove the link itself, but not touch the file MIDAS;MIDAS > linked to. :LINK sets the defaults for future commands from <newlink>, not from <linked-to>. If <newlink> already exists, :LINK will refuse to work, thus preventing accidental deletion of a file. :LINKN is an equivalent command that doesn't check, and can be used to replace a file with a link.

The commands :SFDATE, :SFAUTH, :SFREAP, and :SFDUMP set various attributes of a file: its creation date, its author's name, its don't-reap bit, and its backed-up-on-tape bit. The commands ^T, ◊^T, ◊◊^T, ^U, ◊^U, and ◊◊^U control "filename translations". See the reference section for those commands. The :REAP command marks a file as unimportant, and worthy of deletion as soon as it is backed up on mag tape. There are several commands for handling microtapes: :UINIT, :ASSIGN, :DESIGN, and :FLAP. Since microtapes are hardwarily and softwarily unreliable on the ITS systems, their use is discouraged.
v

File: DDT, Node: Programs, Previous: Files, Up: Top, Next: Jobs

Running Programs with DDT

The simplest way to tell DDT to run a system program is to use the program's name as a colon command; for example, :TECO<cr> will load and start a copy of the text editor TECO. Of course, this will not work for a program whose name is identical to the name of one of DDT's built-in colon commands (such as the DUMP program). DDT looks for the program to be loaded as a file named TS <prgm name>, on any of several disk directories: the user's home directory first, then the several system program directories. The command :NFDIR (which see) can be used to add any other directories to the list.

Once the other program is loaded and started, DDT gives it control of the terminal. Commands typed on the terminal will all go to the other program and not to DDT. DDT will not read any more commands until the program decides to "return to DDT" or gets a fatal error. However, you can at any time tell DDT to seize control and stop the program by typing ^Z (CALL, on TV's). ^Z acts instantaneously, and makes no attempt to be sure that the program has finished acting on the last command you gave it. If you tell TECO to write out your file, it is your responsibility to wait until the TECO says it is done before typing ^Z; otherwise the file may not really exist yet. If you don't want to wait, you might prefer to use ^_D (Control-_ followed by D; on TV's, Control-CALL), which stops the program only when the program "tries to read" the ^_D. Thus, the program will not be stopped until it finishes processing all previous input. So you can type the ^_D in advance, as well as commands for DDT to execute after the ^_D takes effect.

Every user should know about the INFO program for perusing documentation files. Do :INFO<cr> and follow the instructions, which will teach you how to find the topic you are interested in. The INFO program is one of the two places to look for documentation on a program; the other is the .INFO.; directory (See File Manipulation) which contains the older documentation written before :INFO existed.

As described above, every program resides in a job, which is identified by its UNAME and its JNAME. All the jobs created by DDT to run programs in are "inferiors" of the DDT's job; their UNAMEs are the same as DDT's, which was set by logging in, and their JNAMEs are normally the same as the names of the programs that were run in them. Thus, if user FOO runs TECO, it will be loaded into a new job named FOO TECO. He can also do :DDT<cr> to load an "inferior DDT" into the job FOO DDT, and that DDT can then do anything that the top-level DDT in the job HACTRN can do (except log out). The name of a program, such as TECO, is often used as a concrete noun to refer to jobs running copies of it, as in "Now kill your LISPs and start another TECO".

Running one program has no effect on any other jobs the user may have, with other programs in them. Thus, after doing :TECO<cr>, using ^Z to get back to DDT, and doing :LISP<cr>, there will be two inferior jobs: TECO and LISP. LISP will be running, and will have the terminal; TECO will be stopped. The next section of this manual tells how to go back to using the TECO again (with the :JOB and :CONTINUE commands).

When running a program, it can be given arguments by putting them between the program name and the <cr>. For example, :MIDAS FOO<cr> would run the MIDAS assembler and give it the string "FOO" as an argument. Programs treat their arguments in various ways; in this example, MIDAS would assemble the file FOO > and call the binary FOO BIN. Some programs ignore their arguments entirely.

What happens if :TECO<cr> is done when there is already a job named TECO? That question is complicated, because the user has several

options. The best way to explain them is to describe several other commands for running programs, which differ from :<prgm><cr> mainly in what they do in this problematical situation.

The command <prgm>^H will run <prgm> if it isn't already loaded, but simply simply give control back to an existing copy of <prgm> if there is one. It can be thought of as giving control to the program <prgm>, after loading it if necessary. The command <prgm>^K, on the other hand, always loads a fresh copy of <prgm>, and destroys any old copy. For naive users (see ..CLOBRF and :LOGIN), ^K asks for confirmation when it is about to destroy a program. Note that "old copy" really means "anything in a job named <prgm>". TECO^K when there is already a job named TECO will overwrite whatever is in that job, no matter what program it was. :RETRY <prgm><cr> is equivalent to ^K, but makes it possible to specify an argument between <prgm> and the <cr>. Finally, there is :NEW <prgm><cr>, which always loads a fresh copy of the program, but never overwrites any old one! It does this by choosing for the new job a JNAME that isn't in use yet. If there is a job named TECO already, :NEW TECO<cr> will load another copy of TECO into a job named TECO0; if TECO0 too is already in use, it will make a job called TECO1, etc.

So what does :<prgm><cr> itself do when a job <prgm> exists? It acts either like :RETRY <prgm><cr> or like :NEW <prgm><cr>, according to a switch that the user can set (..GENJFL -- see the reference section). Normally, :NEW is chosen, but for naive users :RETRY is done instead, on some machines. That is to prevent them from unwittingly making many copies of programs.

The commands to run programs have features that can be used to load programs from any directory, and to request loading of the program's symbols. The latter is useful mainly when the program is to be debugged. See the reference section "Colon Commands" for full details.
v

File: DDT, Node: Jobs, Previous: Programs, Up: Top, Next: Returning

Job Manipulation

After using the commands :<prgm>, :NEW, ^K, or ^H of the previous section to load programs, one might want to start them and stop them, and eventually get rid of them. DDT has commands for all of those operations.

Jobs in ITS are arranged into trees. Each job is either "top level" or has a specific "superior" job. A job can have up to eight "inferiors"; it is their superior. The HACTRN job is always top-level, and the jobs created by it are its inferiors (unless DDT disowns them - see :DISOWN). DDT is capable of examining any job in the system, but full control (starting, stopping, and depositing in memory) is available only for direct inferiors. Even indirect inferiors (inferiors of DDT's inferiors, etc.) can only be examined.

In order to act on a job, DDT must know which of the up to eight jobs it has it should act on. For brevity of typing, the commands don't say which job. Instead, most of them act implicitly on the job DDT calls the "current" job. The :JOB command can be used to make any of the existing jobs current, so it can be operated on:

:JOB <jname> or <jname>◊J

if DDT knows about a job named <jname>, makes the job current. Otherwise, if such a job actually exists (presumably not a direct inferior of DDT, since DDT always keeps track of those) DDT will henceforth know about it. If it was a disowned job (see :DISOWN), and the top of a tree (disowned jobs too can have inferiors), it is "reowned", which means that it becomes DDT's inferior. In that case, ":◊REOWNED◊" is printed. If there is no job at all with the name <jname> (and the same UNAME as DDT, of course), DDT will create an inferior with that name. A job created in this way has 1K (1024. words) of core (all zero), and no symbols. Its uses might include explicitly loading a program into it, or depositing instructions with DDT and executing them.

Whenever the new current job is not a direct inferior of DDT, DDT types a "#" to tell the user. Whenever :JOB selects a job that DDT didn't already know about, "!" is typed.

:JOB or ◊J

picks a new current job "conveniently". If DDT knows about any jobs other than the current one, one of those others becomes current. Its name is printed out inside an ◊J command, so that the user can see which job DDT chose. Jobs that need attention (are "waiting to return to DDT" because they have received fatal interrupts) are chosen first. Repeated ◊J commands will choose different jobs, and won't return to any job until all the other jobs have been current. Thus, repeated ◊J's are an easy way to visit all of DDT's jobs.

Once a job is current, it can be acted on with these commands:

:CONTINUE or ◊P

makes the current job resume running, and gives it control of the terminal. DDT stops reading commands, and type-in goes to the program in the current job instead. Hardly anyone uses or talks about the long form of this command or the following one, so everyone should know the short forms ◊P and ^P. ◊P undoes the effect of stopping a job with ^Z (remember that ^Z stops a job and makes DDT take back control of the terminal). Thus, after using a program, ^Z goes "back up to DDT", and ◊P can then be used to go "back down into the program". If the current job is already running, but without the terminal (see :PROCEED below), ◊P just gives it the terminal.

:PROCEED or ^P

makes the current job resume running, but DDT keeps control of the terminal. The job's program is not allowed to read from or type on the terminal, but as compensation DDT continues to read commands even while the job is running. If you should change your mind and decide to give the terminal to the program, use ◊P. A ^P'd job will keep running even if DDT no longer considers it current; therefore, ^P can be used to make several programs run at once. If ^P is done on a job that is already running (eg, it has been ^P'd already), it has no effect.

^P is not the only way to make a job run without the terminal, but it is the most common way, so jobs running without the terminal are often described as "^P'd" regardless of how they actually got that way.

◊◊^P

causes the current job to run without the terminal, like ^P, except that it IS allowed to type out. It can't read anything, however; instead, DDT continues to read commands. Use this for a job which you expect will type out briefly and infrequently in the middle of its processing, so you can let it run while using the terminal primarily with another job.

^X

stops the current job, assuming it was ^P'd. ^X is analogous to ^Z -- they both leave a job in the same state -- but they are effective in different situations: ^Z stops a job that DOES have the terminal, while ^X stops a job that DOESN'T have the terminal. The reason that they both exist is that ^Z is an ITS command, that tells ITS to stop whatever job has the terminal, while ^X is a DDT command, and can't be obeyed by DDT if DDT has given the terminal away and isn't paying any attention to it.

:KILL or ◊^X.

"kills" the current job. It ceases to exist, and any data in it is lost. Be sure not to kill a TECO without filing away any information being edited in! The short form ◊^X does not take effect until a period is typed, because it is so dangerous. After killing a job, DDT tries to find a new current job by doing an ◊J without argument. The new current job's name is printed inside an ◊J (select job) command, as in ":KILL TECO◊J", showing that the job TECO is now current (DDT often says what it is doing by printing the very commands that the user could give to request what DDT did).

<job>$^X

kills the job named <job>. It asks for confirmation by typing "--Kill--"; a space tells it to go ahead. Whether you confirm the killing or not, the current job afterward is the same as it was before (unless it was the one you killed).

Although DDT can handle up to eight jobs at once, it is good usership to kill jobs that are no longer needed to free their resources for others. Some programs will commit suicide after finishing their work; when that happens, DDT informs the user by printing out ":KILL " just as if the user had killed the program explicitly.

The :MASSACRE command kills all your jobs.

:LISTJ or ◊◊V

prints a list of all the jobs DDT knows about. Each job gets its own line, which contains the job's JNAME, status, and job number. The

current job is indicated by a "*" at the front of its line. A job's status is usually "P" if it is stopped (proceedable) and "R" if it is running; other states include "-" meaning "never started" (◊P isn't allowed), and W meaning "interrupted and waiting to return to DDT". Example:

```
  TECO P 34
* DEBUG - 25
  SRCCOM R 7
```

TECO is stopped, DEBUG has never been started, and SRCCOM is running. See the reference section for more details. For users on slow terminals, the ◊◊J command prints out just the line describing the current job.

:START or ◊G

starts a job at the starting address of the program in it, unlike ◊P, which starts a job where it last stopped. Many programs do something useful if they are started at their starting addresses after running for a while. For example, a TECO that is hung can be made usable again in that way, without losing any of the data being edited. The ◊G command is useful also after explicitly loading a program into a job with ◊L.

When DDT creates a job, it gives the job the name of your working directory (in the variable .SNAME, which ref). Most programs will use that variable as the default SNAME for files they reference. Of course, many allow the SNAME to be specified explicitly in their commands. The working directory name is known as the MSNAME (for "Master SNAME"), and it is used also for many other purposes, to be described when they are relevant (ref ..MSNAME). For convenience's sake, 0^F is equivalent to <msname>^F; it lists the working directory. The MSNAME can be set explicitly:

:CWD <new msname> or <new msname>◊◊^S

sets the MSNAME to <new msname>. Any jobs created subsequently will be passed <new msname> as the default SNAME to use. Jobs that already exist will not be affected.
v

File: DDT, Node: Returning, Previous: Jobs, Up: Top, Next: Arithmetic

Returning Control to DDT from an Inferior Job

When DDT runs a program, it gives control of the terminal to that program. From then on, DDT expects that your commands are intended for the program instead of for DDT. Three things can change DDT's mind:

The program can get into severe trouble
The program can tell DDT that it is finished
You can insist, by typing ^Z or ^_D.

If any of those three happens, DDT takes the terminal back from the inferior job and resumes reading commands. We say that the job has "returned to DDT" (which does not imply that the return was voluntary). To inform the user of what has happened, DDT prints a description of the job's status and why the it returned to DDT. The conditions that can cause the job to return are called "fatal interrupts", and each one has a name. DDT usually simply prints the names of whichever fatal interrupts happened, but DDT understands some interrupts more and can provide a more hand-tailored response.

When a job returns to DDT, DDT's actions depend on the reason for the job's return. If the job is returning because it requested to do so, DDT simply does whatever the program wanted it to do (but the user can disable this; ref ..PERMIT). In this case, DDT is likely just to print a "*" indicating completion of a command, or ":KILL " indicating that the job decided it was done or useless, and DDT got rid of it. If the job returned because it tried to read past a ^_D, DDT just types a "*". In other cases, the return is regarded as a sort of error in the program, and DDT prints a message. DDT's message usually contains the job's PC and the next instruction it will execute if ◊P'd. This information is provided for debugging's sake, and you should not be worried by it.

```
105)   JRST 105
```

is the sort of message DDT prints when a program is stopped with a ^Z (or a ^X!).

```
ILOPR; 0>>   0
```

indicates that the program ran into trouble: namely, an ILOPR (illegal operation, in this case executing a zero). In general, the ">>" indicates that the program encountered an error, and the type of error is named. A complete list of error condition names can be found in the reference section "Returning to DDT".

Instructions which cause errors are usually "aborted", which means that any side effects are undone and the PC is left pointing at the instruction itself (instead of the next one). As a result, the instruction printed by DDT is not only the next to be executed, but also the one responsible for the problem. An unfortunate exception is the PDLOV (stack overflow or underflow) error, which does not abort the instruction; either the offending instruction is the one before the instruction printed, or the .JPC (address of last jump instruction) points to it.

If you see other names in parentheses, as in

```
ILOPR; (REALTM;) 0>> 0
```

they are names of other non-fatal interrupts which are pending for the job. They are printed for debuggers' convenience, and have no necessary relationship to the reason the job returned.

If a program has been ^P'd and is running without control of the

terminal, that doesn't stop it from getting into trouble or finishing, and then trying to return. But since the terminal is being used for DDT or some other program, DDT doesn't allow the program to return just yet. Instead, the program has to wait, and in a :LISTJ or ◊◊V its status will be "W" for "waiting". DDT announces this development to the user with a message such as "Job <jname> interrupting: ILOPR", "Job <jname> finished", or "Job <jname> wants the TTY", or something else similar. When an ◊J with no argument is done, the job will be allowed to return, and DDT will print the appropriate message. "Job <jname> wants the TTY" indicates that the job tried to read from or type on the terminal when the terminal belonged to DDT or some other job. "... finished" means that the program has finished its task and has asked DDT to kill its job. In some cases, the job will in fact be killed at the same time you receive the "finished" message.
"... interrupting ..." means that the job has encountered an error condition, whose name is printed.
v

File: DDT, Node: Arithmetic, Previous: Returning, Up: Top, Next: Sophisticated

Arithmetic in DDT.

DDT's role as a debugging aid requires that it be able to serve as a desk calculator. Expressions involving numbers (fixed or floating point) and arithmetic operators such as +, -, etc. are evaluated, and can be used as arguments to appropriate commands. One command that makes it easy to learn how expressions in DDT behave is the "=" command, which prints the value of the expression preceding it.

For convenience in debugging a machine which uses binary arithmetic, numbers input to DDT are normally interpreted as octal. However, if a number is followed by a decimal point then it is (naturally) treated as decimal. Thus, 12 and 10. are equal. A number which has a decimal point in the middle or at the front is interpreted as floating point. Floating point numbers are always read as decimal, and can end with "E" followed by an exponent of ten, as in 1.1e5 which is the same as 11000.0 . These conventions NEVER vary.

DDT has separate operators for fixed point arithmetic and floating point arithmetic. This is because once a number has been read in DDT does not remember which one it was. There is no way for DDT to know whether two numbers must be added using fixed point arithmetic or floating point arithmetic, so the user has to specify one or the other for every arithmetic operation. In this matter DDT resembles the machine language that it was intended to debug, rather than high level languages. Specifying the mode is not difficult, however, because the floating point arithmetic operators are just the fixed point operators with a single extra altmode. Thus, floating point addition is done with ◊+, and floating point multiplication with ◊*. The = command always prints its argument as a fixed point number. Its floating point equivalent, ◊=, always prints its argument as a floating point number. Feeding fixed point numbers to the floating point arithmetic operators or ◊=, and the reverse, will produce seemingly insane results. This are a good way to learn about the PDP-10's floating point number representation and the quirks of its floating point arithmetic instructions.

Here is a list of all DDT arithmetic operators:

+	fixed addition	◊+	floating addition
-	fixed subtraction	◊-	floating subtraction
*	fixed multiplication	◊*	floating multiplication
!	fixed division	◊!	floating division
#	bitwise exclusive-or		
&	bitwise and		
◊_	logical shift	◊◊_	floating scale

The &, # and _ operations are done first, multiplication and division second, and addition and subtraction last. The subtraction and division operators may be unary, as in "-50" which has the obvious meaning. Unary fixed point division is rather useless, but "◊!2.0" is 0.5. The logical shift and floating scale operations are defined by the PDP-10 instructions LSH and FSC. They LSH or FSC their first operands by a number specified by their second operands.

DO NOT use extra spaces inside arithmetic expressions. They are not ignored and will alter the value. This results from the PDP-10 instruction type-in features, to be described later.

Terms in expressions can include symbols and some special quantities, as well as numbers. Symbols always have numeric values if they are defined at all, so they are used just like numbers. Since they are useful mainly for debugging, their detailed description is postponed.
v

File: DDT, Node: Sophisticated, Previous: Arithmetic, Up: Top, Next: Loading

Sophisticated Job Manipulation

This section describes other things that DDT can do with jobs, that are less often useful or more difficult to understand than those in the previous section.

:UJOB <uname> <jname>

is somewhat like :JOB. It is used to select some other user's job, usually for examination only. It makes it possible to specify the UNAME of the job to be selected instead of just the JNAME. When a :UJOB is done, there are three major possible situations and outcomes:

1) User <uname> doesn't exist, or has no job named <jname>. In that case, :UJOB doesn't create a job -- it is an error.

2) The specified job exists and is disowned. In that case, the job is reowned.

3) The job exists and is not disowned. In that case, it is selected for examination only.

Once another user's job has been made known to DDT with :UJOB, just plain <jname>◊J can be used to make it current again. That ◊J will print :UJOB <uname> <jname> to show you what is happening. The same thing will be printed by a ◊J with no argument, if it selects the foreign job. Commands to run programs, such as ^K, ignore totally any jobs whose unames differ from yours; they go ahead and create an inferior in addition to the unsuitable job.

:DISOWN or ◊◊^K

"disowns" the current job. The job continues to exist, and if it was running continues to run, but it ceases to be DDT's inferior. Any information DDT has about the job that is not actually in the job itself is lost (for example, the starting address and symbols of the program). When a job has been disowned it no longer has a terminal, and if it tries to read from or print on its terminal it will halt. Disowning allows a job to continue to exist after the DDT that created it has logged out or been killed. It makes it possible to leave a job running without tying up a terminal.

A disowned job can be reowned by selecting it with :JOB. What's more, any user can reown a job no matter who disowned it, using the :UJOB command and specifying the UNAME of the disowned job, as in :UJOB FOOSH TECO to reown the TECO that user FOOSH disowned. This makes it possible to hand a job to another user.

:ATTACH

makes the current job (which must be running) become the top level job, in place of DDT. That job's name is changed to HACTRN, and the existing HACTRN job (containing DDT) is killed, along with any other inferiors it may have. :ATTACH is very dangerous for that reason. Its main use is to set up a program other than DDT as the top-level command processor. It is possible to use :ATTACH to do the opposite of :DETACH. Just reown the detached former HACTRN (now called HACTRO or HACTRP or ...) using :JOB, and then :ATTACH it. However, it is probably safer to use the REATTACH program. Type :REATTACH ?<cr> for information.

:FORGET

makes DDT forget that the current job exists. The job is untouched, and even remains DDT's inferior, but DDT no longer knows about it. Why do this? So that DDT will no longer mention the job,

and :MASSACRE won't kill it, but the job will remain in your tree (so it can type out on the terminal if it has been ◊◊^P'd).

:SNARF <jname>

When a HACTRN is detached because of trouble with the terminal, but is still basically healthy, it can be attached. When a HACTRN is detached because of fatal errors, it stops running and can't be attached (and, having run into such trouble, it would probably be useless if it were attached). However, its inferiors are likely to be unharmed. The :SNARF command exists to rescue those inferiors from under the sinking DDT. It is meant to be used after reowning that DDT as a subjob of a new, healthy DDT. The dead DDT should be the current job. :SNARF takes away the current job's inferior named <jname> and makes it a direct inferior of the DDT executing the :SNARF. Thus, after a HACTRN dies while having a TECO under it (and thus changes to a HACTRO), one can do (in a new HACTRN) :JOB HACTRO to reown the dead DDT, and :SNARF TECO<cr> to take the TECO away from it. The job TECO is then an inferior of the new HACTRN, and the HACTRO job can be killed without harm to the TECO. If you try to :SNARF a nonexistent job, a "No such job" error will result. :SNARF works by writing into the current job a program to disown any inferior named <jname>, and then doing a :JOB <jname>. Thus, :SNARF can garbage the job snarfed from. This is small loss when the job is already dead.

<newjname>◊◊J

changes the current job's name to <newjname>. The job's contents are unchanged, and so is what DDT knows about it; only the name is changed. Another command, :GENJOB, changes the job's name to a "generated name" chosen not to conflict with any other job.

:GZP<cr>

starts the current job at its program's starting address, without giving it control of the terminal. :GZP is like an instantaneous sequece of ◊G (:GO), ^Z, and ^P, which is how it got its name.

:JCL<cr>

clears out the current job's comand buffer. <prgm>^K clears the command buffer of the job it creates.

:JCL <line of text>

puts <line of text> in the current job's command buffer. <line of text> must end with a ^C or <cr>. :<prgm> <text> does a :JCL <text> to the job it creates. Note that the command buffer is actually stored inside DDT. Programs use it by reading its contents with a .BREAK instruction, and :JCL cannot retroactively alter what previous .BREAK instructions have read.
v

File: DDT, Node: Loading, Previous: Sophisticated, Up: Top, Next: Communication

Loading and Dumping Jobs

:LOAD <file><cr> or ◊L <file><cr>

loads the binary file <file> into the current job, after resetting it. Resetting a job destroys all its core and reinitializes most of its system variables (filename translations are the main exception). Symbols are forgotten and replaced by those loaded from the file. Breakpoints, raid registers and JCL remain set. ◊L does not use the same set of default filenames that the "File manipulation commands" above use; instead, each job has its own default filenames that ◊L and all other loading and dumping commands use when acting on that job. When a job is created by ◊J (:JOB), its loading default names are initialized as DSK:<msname>;<jname> BIN. Thus, the easiest way to start debugging the file TECO BIN is to do TECO◊J ◊L<cr>. When a job is created by a program-running command, the loading and dumping filename is set to the name of the program run. You can specify a list of directories for ◊L to search for files in using :NFDIR (ref :NFDIR). For more information, see the reference section under "Defaulting of Filenames", and under ◊L. Some very old binary programs may not load with ◊L and will require the :OLOAD command (ref :OLOAD).

◊0L <file>

loads a core-image file instead of a binary file. Of course, that's a matter of a different interpretation of the file, since files do not say that they are binary or core image. What actually happens is that the file's contents are copied directly into the memory of the job. This makes it possible to use DDT's debugging commands to examine the contents of the file. The current location address (".") is set to the first word not loaded, to make it possible to find the end of the data easily.

The ◊L command has two other options, which may be used with either ◊L or ◊0L. Two altmodes instead of one (◊◊L or ◊◊0L) cause a merge-load, which does not throw away the core and symbols the job already has. The data in the file replace the data in core, but locations not loaded by the file are unchanged. Symbols already defined are kept, along with any symbols in the file. Also, the job's system variables are not reinitialized. An infix 1 (◊1L or ◊◊1L) loads a binary file without its symbols.

<addr>◊L <file>

loads with an offset of <addr>. A binary file has <addr> added to all the addresses it specifies loading into (unfortunately, addresses in the program cannot be relocated). A core image file (<addr>◊0L) is simply read in starting at <addr>.

:DUMP <file> or ◊Y <file>

dumps the contents of the current job's core as an SBLK file.

:PDUMP <file>

dumps the contents of the current job's page map as a PDUMP file. SBLK and PDUMP files have different advantages and disadvantages. SBLK files remember the contents of the job's accumulators, and take up less space because (when dumped by DDT) they are zero- compressed. Zero-compression means that the contents of each nonzero location is recorded; nothing is said about locations containing zero. Because of this, the size of the file varies with the amount of nonzero data in it. Zero-compression facilitates merging programs, since loading a zero-compressed file alters only the locations specifically mentioned in the file. The disadvantage is that a zero-compressed file does not

distinguish between memory that is all zero and memory that (virtually) does not exist at all. PDUMP files remember the entire state of the page map. Thus, they record which pages are read only, and record gaps of non-existent memory between existing regions, as well as mapped system pages and pages shared between two slots in the address space. All of that information is used to reconstruct the page map when the file is loaded. The most important use of PDUMP files is for sharable system programs, because when a PDUMP file is loaded all read-only pages remain shared with the file, and will therefore be shared between all jobs that load the file. Both kinds of binary files will contain the job's symbol table and start address.

The ◊Y command provides other types of dumping operations:

◊0Y <file>

writes the core of the current job directly into <file>, as a core-image. If the job has 5K of core, the file will be exactly 5K long.

<low>◊,<high>◊Y <file>

dumps, as an SBLK file, the range of core from <low> to <high>, inclusive. Other core locations in the job's memory will simply not be mentioned in the SBLK file and will not be altered if the file is loaded.

<low>◊,<high>◊0Y <file>

writes out words <low> through <high> of the current job into <file> as a direct core image. Exactly <high>-<low>+1 words are written.

Related commands include :LFILE, which prints out the name of the last file loaded (not necessarily the same as the current ◊L default file), and ◊◊Z, which sets all or a specified range of the job's memory to a specified value (usually zero).
v

File: DDT, Node: Communication, Previous: Loading, Up: Top, Next: Announcements

Communication with Other Users

The three forms of inter-user communication commonly found on various timesharing systems all exist on ITS. They are known as "sending", "linking", and "mailing". In sending, you compose a message, and when finished cause it to appear all at once on the other user's terminal. Linking (not to be confused with file links or :LINK) puts two or more users into a "com link", after which any character typed by any of the users appears on all of the linked terminals. Mailing writes a message that another user will see when he next logs in; unlike sending and linking, it does not require that the other user be logged in when it is done. Com links are good for carrying on a conversation, especially a many-way one, but they have the disadvantage of interfering more with doing work at the same time. Com links really have nothing to do with DDT, since they are implemented by ITS directly. For information on them, see the file .INFO.;ITS TTY.

:SEND <user> <message>^C

sends <message> to <user>. Nothing actually happens until the ^C is typed, and until then the command can be cancelled with ^D and individual characters can be cancelled with <rubout>. The message can be any number of lines long. When the ^C is typed, <user> will see printed on his console:

```
MESSAGE FROM <sender> HACTRN
<sender>@<machine> <time of day> <message>
```

<sender> is your UNAME, and <machine> is the name of the machine you are on (AI, ML, MC or DM). <machine> is included in case <user> is using one machine from another over the ARPA network; he needs to know which machine he got the message from.

All the messages you are sent will be put in your "SENDS file", COM:<your uname> SENDS, which is deleted when you log out. If you miss a send because it is overwritten on the screen, just print that file to see it. In addition, DDT will print a message repeatedly if it is afraid you are likely to have missed it; when you have seen it you can stop the repetition by typing ^Z (ref ..SENDRP).

:PRSEND<cr>

prints out your SENDS file.

:PRSEND <user><cr>

prints out <user>'s SENDS file.

After just :SEND <user><space> has been typed in, DDT checks whether <user> is logged in at the moment. It is impossible to send to a user who isn't logged in. For convenience's sake, DDT turns the :SEND command into a :MAIL command, and types (MAIL). Since :SEND and :MAIL have almost the same syntax, it usually isn't necessary to pause for this. If <user> was logged in at the beginning of the :SEND, he might still log out while the message is being typed in. In that case, DDT automatically mails the message instead of sending it (and types out (MAIL)).

There is also a program QSEND that can be used to send. It is less efficient than :SEND, but it can send to more than user at once, and can send to users logged in on other machines. QSEND is really the same program that does mailing, and its documentation is in .INFO.;MAIL ORDER.

There are times when it would be embarrassing to receive a message

(for example, when printing a copy of this file). For those times, :GAG is available.

:GAG 0

tells DDT not to accept messages. If anyone tries to :SEND to you, his DDT will do :MAIL instead.

:GAG 1

tells DDT to resume accepting messages.

To be completely certain that a printout won't be garbaged, :GAG is not enough. For one thing, it is necessary to refuse com links with ^_R (see .INFO.;ITS TTY). In addition, to stop DDT from notifying you of various things, the :NOMSG command can be used. It looks like :GAG, but controls a different switch which is more powerful. :NOMSG 0 won't bother you with anything except an emergency (ITS going down in less than 15 minutes).

In emergencies, such as when the disk is almost full, it may be necessary to send to all users at once. The :SHOUT command does that -- see the reference section.

To communicate with a user who is not currently logged in, :MAIL must be used. :MAIL is not actually a DDT command; it runs the MAIL program. :MAIL has many features; for example, it is easy to mail to several users, on any ARPA network hosts. For full details, see *note MAIL: (INFO;MAIL >). The simplest usage, though, looks exactly like :SEND:

:MAIL <user> <message>^C

There is also a DDT command :OMAIL, which is the operation that used to be called :MAIL, before the more general MAIL program existed. It is kept as a backup for the MAIL program. Its syntax is exactly like :SEND's.

:BUG <prgm> <message>^C

mails a complaint about the program <prgm> to the "appropriate" person(s). If you couldn't figure this out, you might complain by doing :BUG DDT IN DDT DOC, THE DESCRIPTION OF :BUG IS UNCLEAR^C. :BUG actually invokes the MAIL program, and is equivalent to :MAIL BUG-<prgm> <message>^C. Thus, :BUG DDT... mails to BUG-DDT. From there, the mailer "forwards" the message to the people who have asked for that. If the name is not recognized, the message goes to the system maintainers (BUG-RANDOM-PROGRAM), so every :BUG message is guaranteed to be seen by someone. If a program fails to do what it is supposed to do, please report it; bugs have sometimes existed for months, known to everyone except the person who could have fixed them. But first ask your neighbor to verify that you aren't simply confused, or scrod by obsolete documentation.

There are several ways to read mail you have received. Normally, mail goes in a file <uname> MAIL, which is on the directory <uname>; if it exists, or on COMMON; otherwise. Knowing that, you can use :PRINT to read it. In addition, there is a special command to do that:

:PRMAIL<cr>

prints out your mail file, and renames it to <uname> OMAIL. As a result, each :PRMAIL shows only new mail arrived since the previous :PRMAIL. The OMAIL file is deleted when you log out, or when you do another :PRMAIL command.

:PRMAIL <user>

prints out <user>'s mail file, without renaming it or otherwise altering anything. :PRMAIL <self> is good for looking at your own mail without renaming it.

Normally, DDT does a :PRMAIL automatically when you log in. If you have a DDT INIT file, no :PRMAIL is done unless the INIT file calls for one.

There is also a sophisticated program for reading, answering and distributing mail, called RMAIL. It is intended primarily for display terminals, but can be used from printing ones. Run :INFO and look under RMAIL to see its documentation. If you use RMAIL, you won't want DDT ever to rename your mail file to OMAIL, but you might still want to do :PRMAIL once in a while for a quick glance at your mail when you don't want to edit it. Putting the commands :DDTSYM OMAILF/ 0<cr> in your init file will make :PRMAIL<cr> just print your mail file, without renaming it.

One other user you might want to comunicate with is yourself at another time. The :ALARM command tells DDT to notify you when a specific time arrives:

:ALARM <hour>:<minute>

specifies an alarm. The argument is the time of day in 24-hour form and is absolute, not relative to the time the command is given. As soon as DDT reads the command it will print

Alarm set for .+<duration as hours:minutes>

telling how far away the specified time is. If it says "Alarm reset" instead of "Alarm set", there was a previously specified alarm in effect. That alarm is forgotten, as DDT can remember only one alarm at a time. When the alarm comes due, DDT will begin printing a message on the console frequently until the alarm is explicitly cleared.

:ALARM<cr>

clears any alarm, whether it has come due already or not.
v

File: DDT, Node: Announcements, Previous: Communication, Up: Top, Next: XFILE

Announcements

Messages of general interest to the user community, but not urgent enough to be printed out by DDT when it starts up, are distributed as "announcements". Announcements are really just disk files, and can be :PRINTed, but they are normally read with a special command, :MSGS, that contrives to show any given user each announcement exactly once. It does so by keeping track, for each user, of the creation date of the most recent announcement he has seen; only announcements more recent than that are eligible for being printed. The date information is kept in the file called _MSGS_ <uname> on your directory, so don't delete it.

:MSGS<cr>

prints out any recently created announcements, that you haven't seen before with :MSGS. If there is a new announcement, DDT prints --MSGS-- to ask if you wish to see it (yes, even though you just asked to see them! The reason will become clear). If you answer yes, DDT prints the filenames and first line only of the announcement, and then asks with --More-- whether it should print the rest. If you say "yes" (with a <space>), the rest of the announcement will be printed. In either case, after finishing with one announcement, DDT checks for other new announcements, and if there are any the whole cycle repeats. The announcements are offered in forward chronological order. If at any time you answer "no" (with <cr>) to --MSGS--, DDT remembers the date of the last announcement it printed, and says "Deferred" to indicate that the remaining announcements will not be printed now, but will be offered again by the next :MSGS command. The command and offer are "msgs" because announcements used to be called "messages"; that, however, unfortunately led to much confusion with mail.

The --MSGS-- offer is unlike all others in that <rubout> means "yes", just like <space>. Any other character still means "no". This is so that by typing nothing but <rubout>s one can see the filenames and first lines of all the announcements.

Normally, DDT will do a :MSGS for you automatically when you log in, if you either have a file directory or have ever done a :MSGS before. However, if you have a DDT init file, to be executed when you log in, this default is overridden; it is the init file's responsibiliy to do a :MSGS if that is desired. An alternative to :MSGS is the program :GMSGS, which copies any newly created announcements into your mail file. You can then read the announcements along with your mail, using RMAIL or any other mechanism. See .INFO.;GMSGS ORDER for information on GMSGS. Because :MSGS and :GMSGS remember the date of the most recent announcement seen in the same place, it is possible to switch between :GMSGS and :MSGS without losing track of anything.

:MSGS <keyword1>,<keyword2>,...

A fact that is normally invisible is that each announcement contains one or more "keywords" which indicate which machines' user communities the announcement is intended for. For example, an announcement might be intended for MC and ML only. It would then have the two keywords "*MC" and "*ML", and as a result it would then normally be seen only by users on MC and ML. But in fact it would be present on all ITS systems, and a user on AI who wished to see it could do so, by specifying explicitly in his :MSGS or :GMSGS command the keywords he is interested in. Showing only announcements intended for the machine you are running on, is just the default. An example of a keyword argument is *AI, meaning "show messages intended for the AI machine". * as an argument means "show all messages, no matter what machine they are intended for". People who wish to stay abreast of all developments in the ITS community do :MSGS *<cr>.

Since there is only one remembered date for announcements, instead of one for each keyword, announcements can be missed if you do not consistently use one set of keywords in every :MSGS or :GMSGS you do. For example, if your last :MSGS was a :MSGS * three days ago, and you do :MSGS<cr>, you will see any announcements intended for the machine that you are on, created in the last three days, and your remembered date will be updated to the current date. As a result, if there were any announcements NOT intended for the machine you are on, created in the last three days, you will not be shown them by :MSGS *, since it will assume you have seen them.

Announcements are submitted with the MAIL program, by using the desired keywords as the "recipient names". Thus, :MAIL *AI<cr> would create an announcement with the single keyword *AI. :MAIL *ITS<cr> creates an announcement with all four keywords *AI, *DM, *MC and *ML - this announcement will be seen by everyone on all four machines. The MAIL program will request all appropriate information such as the expiration date of the announcement, the desired file name, and the subject, which will appear on the first line of the announcement. Most announcements should expire in 7 days, but announcements of changes in system programs should expire in 30 days. Of course, announcements that are of no interest after a specific day should expire then.
v

File: DDT, Node: XFILE, Previous: Announcements, Up: Top, Next: Symbols

DDT Commands from Files or Programs

Although DDT normally reads its commands from the terminal, it can be told to take them from a file (called an "execute file" or "xfile"), and a running program can order DDT to execute a specific string of commands (The operation is called "valretting" and the string is a "valret string"). Although for the most part commands in execute files and valret strings are written exactly the same way they would be typed, there are a few exceptions. This section describes those exceptions, as well as some commands that are useful primarily in files or valrets.

The most common use of execute files is as init or exit files, which are executed automatically (if they exist) upon logging in or out, respectively (but ◊0U and ◊◊0U allow you to log in or out as if you had no init or exit file). Init files are identified to DDT not by a user command but by having the appropriate filenames. An init file is called .DDT. (INIT) on the <xuname>; directory, or .DDT. <xuname> on the DDTINI; directory. An exit file has .DDT_ instead of .DDT. in its name. DDT init and exit files are expected to be on the DDTINI; directory instead of the (INIT); directory, where other programs look, to prevent (INIT); from becoming full.

However, the user can explicitly command the execution of a command file at any time:

:XFILE <file>

tells DDT to begin reading commands from <file>. :XFILE has its own set of default filenames, not shared with any other command. Thus, :XFILE<cr> is guaranteed to re-execute the last file :XFILE'd. The initial defaults are .DDT. (INIT). When the end of <file> is reached, DDT will resume reading commands from the terminal. Until then, DDT will read no input from the terminal. Interrupt-action characters such as ^G and ^S will still have effect, but any other input will not be read until the file is finished (unless the file runs a program which reads the input). ^S-silencing stops only when the ^S is read, so if done while an execute file is running the whole execute file will be silenced.

Valret strings are given to DDT by the execution of a .VALUE instruction in an inferior job. .VALUE is described in the section "DDT Services for Programs Running Under DDT".

Note that the execute file affects only DDT; it does not also supply input to other programs. Programs that normally read input from the terminal will continue to do so, even when invoked by an execute file. However, command strings of programs (a la :<prgm> <command>) are really read by DDT and can be specified by execute files.

Normally, the commands read from an execute file will be typed on the terminal as they are executed, making the typescript appear as if the user had typed the commands on the terminal when it was time. However, the file can use the characters ^V and ^W to turn off output to the terminal, and that affects echoing of the commands as well. In fact, most execute files and valret strings start with a ^W and end with a ^V, so that they print nothing at all when they execute. The characters ^V, ^W, ^B and ^E in execute files are interpreted only when DDT has finished handling everything before them. This is unlike the way they are treated when typed on the terminal; then they are interpreted immediately when typed, even if DDT is still processing previous input. Also, some of them have slightly different effects in an execute file, to make programming more convenient. ^W-^V pairs in files can be nested, and nothing is printed on the terminal if it is within at least one ^W-^V pair. See the reference section.

Since files almost always have a ^L at the end, DDT ignores the character ^L in execute files and valret strings. To clear the screen, the :CLEAR command must be used. DDT commands are often terminated by stray <cr>'s, but stray <cr>'s look ugly in files and in assembly sources. So in files and valrets DDT allows each <cr> to be followed by a <lf>, which is ignored. If the character sequence <cr> <lf> is actually intended, the file must contain <cr><lf><lf>; only the first <lf> is ignored.

The character ^C is the traditional end-of-file indicator on ITS, so any ^C will be treated as the end of the file. This may change with planned improvements in the ITS file system. ^C does not terminate valret strings; they should be in ASCIZ format.

Execute files and valret strings work recursively. That is, one execute file or valret string can do a :XFILE of another execute file, or run a program that submits another valret string. When that happens, the inner file or string is executed, and at its end the execution of the outer one resumes.

Errors during the execution of the commands in an execute file or valret string do not irrevocably prevent the execution of the rest of the file or string. Instead, DDT postpones or "pushes" the rest of it (typing ":INPUSH " to inform the user), and starts taking commands from the terminal again. The user can recover from the abnormal situation and then cause the rest of the file or string to be executed, with the :INPOP command:

:INPOP<cr>

causes DDT to resume executing commands from the most recently suspended execute file or valret string.

:INPOP <line><cr>

tells DDT to resume the file or string, after executing the DDT commands in <line>. For example, an abnormal return from an inferior job counts as an error and suspends execution of the file or string. After fixing up the problem with the program, one might restart the program and resume execution of the file's commands by doing :INPOP ◊P<cr>.

If the error shows that the rest of the file is not wanted, the command :INFLS will discard all the suspended files and valret strings. ^G has the same effect. It is wise to do this because DDT is limited in the depth to which it can nest execute files and valret strings; leaving unwanted ones stacked can interfere with normal operation later. When the maximum nesting depth is exceeded, the error message "INPDL OVERFLOW" results and all the suspended files and strings are discarded. Note that a :XFILE at the very end of an execute file or valret string does not use any extra stack space; the file that is ending is popped before the new one is pushed.

Complicated programs can be written for DDT, since execute files allow both conditionals and loops.

Conditionals use the :IF command:

```
:IF <condition> <argument>
◊( <conditionalized commands> ◊)
```

executes <conditionalized commands> or does not execute them, according to <condition> and <argument>. The simplest conditions are the numeric sign conditions: L, E, LE, N, G, GE. Those conditions expect <argument> to be a numeric expression and test the sign of its value. Thus, :IF E 1 would fail and not execute the <conditionalized commands>. <conditionalized commands> must be balanced in parentheses - that is how DDT knows when they end. The ◊(and ◊) commands are ignored by DDT except for their effect on the parenthesis counting, so

commands containing unbalanced parentheses can be conditionalized by including an extra ◊(or ◊) to balance the string. Conditionals are strong enough to affect even the output-control characters ^V, ^W, ^B, ^E. Thus,

```
^W :IF N ◊Q
◊( :◊^VThis is a printed message
^W◊
◊)^V
```

conditionally prints "This is a printed message<cr>". The <cr> before the ◊) is necessary to end the :◊ ... ◊<cr> comment construction. The :IF argument can be ended by ">" instead of <cr>, if the user wishes.

:EXISTS <file><cr>

returns 0 if <file> can be successfully opened for reading. If it can't be, the I/O channel status containing the error code is returned (it will be nonzero). Thus,

```
:EXISTS FOO
=
```

would print 0 if FOO exists. :EXISTS is very useful in arithmetic conditionals:

```
^W :IF E :EXISTS FOO NOTE
> ◊( :PRINT FOO NOTE^V
^W ◊) ^V
```

prints the file FOO NOTE if it exists. The ^V<cr>^W construct causes typeout to be enabled when the :PRINT is executed - otherwise, the file would not actually be printed! The <cr> after NOTE ends the :EXISTS. It is used up thereby, and does not end the argument to :IF (you are allowed, for example, to have an arithmetic operator there). The :IF argument is ended by the ">", although a second <cr> would have done just as well.

If you want to have an else clause in your conditional, use the :ELSE command.

```
:ELSE
◊( <commands> ◊)
```

is a conditional which succeeds if the preceding conditional failed. The "preceding conditional" is the last one which ended; it may have contained others, but they don't matter. Successive :ELSE's will alternate between success and failure.

:ALSO is like :ELSE, but succeeds if the preceding conditional SUCCEEDED.

A common thing for a conditional to do is to examine the contents of a location in DDT itself. Many DDT locations are useful for conditionals, but are not useful enough to have user-visible symbols that refer to them. The :DDTSYM command makes it easy to open such locations:

:DDTSYM <symbol>

is the value of DDT's internal symbol <symbol>, considered as an address inside DDT. That is, if the :DDTSYM is used as the argument to, for example, the / command, the location in DDT will be opened. Useful DDT locations include TTYTYP, TTYOPT, and TCTYP, which hold the values of the terminal's system variables with the same names. They are useful for conditionals in init files designed to turn on various terminal options according to the type of terminal or whether it is local or not. See .INFO.;ITS TTY for details of what these variables contain. Some people who usually log in remotely

from a particular type of terminal have tried putting :TCTYP's in their init files. When those people visited the lab and logged in on a local terminal, the :TCTYP lied to the system, making the terminal appear to be broken. In addition, if you use the AI, MC, ML or DM program to communicate with another machine, and do a TCTYP to change the terminal type, you will screw up your connection. Here is how to conditionalize the :TCTYP which sets the parameters for your home terminal so that it is done only on remote terminals:

```
:DDTSYM TTYTYP/                 (this tests for STY or dialup)
:IF N ◊Q&<%TYSTY+%TYDIL>
◊( :DDTSYM TCTYP/               (this makes SUPDUP terminals
   :IF N ◊Q-%TNSFW               not count)
   ◊( :TCTYP LINEL 69. etc.
◊)◊)
```

To test, in a conditional, whether a symbol is defined, the command :SYMTYP can be used - see the reference section.

:IF MORE 0

is another type of conditional, which asks for input from the user. If the user types <space>, the conditional succeeds; otherwise, it fails. If the user types anything but <space> or <rubout>, the character is left around to be seen later. These conventions are identical to those of --More-- and all other unsolicited offers built into DDT. Thus, :IF MORE lets the user put his own unsolicited offers into execute files. The "0" is there simply because :IF always requires a numeric argument; at the moment, its value is ignored. The following code declares the terminal to be a tektronix, if the user says <space>.

```
^W :◊ ^V--Tektronix--^W◊
:IF MORE 0
◊( :TCTYP TEKTRONIX
◊) ^V
```

:MORE <line>

is a simpler but less versatile way of asking the user a question. It prints <line> on the terminal, and then does a :INPOP (exiting the execute file) unless the user answers space. :MORE automatically does as many ^V's as are necessary to make the line actually appear on the terminal.

General transfers of control are available in execute files and valret strings with the :TAG and :JUMP commands.

:JUMP <tag>

transfers control to the specified tag. :JUMP is normally allowed only in execute files and valret strings, and the tag must be defined in the same file or string. If a tag is defined more than once, the first definition is always the one that is found. Nonlocal :JUMPing is not allowed, with one exception: if DDT is reading from the terminal after :INPUSH'ing a file or valret string, you can :JUMP to a tag in that file or string.

:TAG <tag>

defines the tag <tag>, for :JUMP's to refer to. :TAG is a no-op when it is encountered in the normal sequence of execution.

:SLEEP <30'ths>

waits (doing nothing) for <30'ths> 30'ths of a second. This command is useful for execute files that loop, doing something at fixed time intervals:

```
^V :TAG LOOP
<do something>
^W :SLEEP 5.*30.
:JUMP LOOP
```

Init and exit files often want to perform the normal actions in addition to the particular actions which make the init or exit file necessary. They can use the :INTEST and :OUTTEST commands, which perform DDT's default logging-in or logging-out actions (what DDT does if there is no init/exit file). Beware: :INTEST has that meaning only when used in an execute file! See the reference section. :MSGS and :PRMAIL are also likely to be useful in init files. The programs GMSGS and RMAIL offer alternative ways of reading messages and mail - see their documentation files on .INFO.;.
v

File: DDT, Node: Symbols, Previous: XFILE, Up: Top, Next: Memory

Symbols

Symbols in DDT are used primarily for debugging, and to better serve that purpose they do not work as they might in a typical interpreted programming language. In DDT, all defined symbols are either predefined system symbols that are always available to all users, or job-specific symbols that are associated with a specific program, and were (probably) loaded along with the program. Thus, the meaning of a symbol in DDT while debugging a program is about the same as the meaning it has in the assembler when the program finished assembling. DDT's predefined symbols, for the most part, are the same as MIDAS's predefined symbols; they include all the PDP-10 instructions, all the UUOs of ITS, and many quantities useful as arguments to ITS system calls. Of course, assembler macros and pseudo-ops will not be known to DDT at all.

The syntax of a symbol in DDT is the same as that in MIDAS: anything made of letters, digits, and ".", "$", and "%", which does not make sense as a number, is a legitimate symbol, and only the first six characters of a symbol are significant. Note, however, that the set of reasonable numbers in DDT is not the same as in MIDAS, since DDT uses "E" to signal a floating point exponent, while MIDAS uses "^".

Almost any symbol defined in DDT at all is defined numerically. Therefore, symbols are used exactly as numbers are used. Just as in MIDAS, the symbol "." is special; in DDT it refers to the address of the last location examined.

<symbol>:

defines <symbol> to equal "."'s current value. <symbol> is defined only for the current job.

<expression> <symbol>:

defines <symbol> to have the specified value.

In limited contexts, DDT allows you to make "forward references" to a symbol which you are going to define later. This makes DDT in effect a complete one-pass assembler. Forward references may be used only to deposit into memory, and only in the address field of a word or the left half. Furthermore, aside from adding a known value to the forward reference, no arithmetic is allowed.

<symbol>?

is a forward reference to the symbol <symbol>. Thus, "MOVE A,FOOO1?<cr>" will deposit a MOVE instruction referring to the as-yet-undefined address FOOO1. When FOOO1 is later defined with "FOOO1:", the new value will be stored into the MOVE instruction. Until then, the job's "undefined symbol table" will contain an entry indicating that the value of FOOO1 must be added to the location in which the MOVE instruction was stored. The undefined symbol table is loaded and dumped along with the regular defined symbol table; its contents can be printed with the :LISTU command.

When you type in a reference to an undefined symbol, DDT does not detect that until the following operator is typed in. This is because that operator tells DDT how to interpret the symbol (imagine what happens if the operator is :, or even ^F). When DDT does realize that you have typed an undefined symbol, the special error message "?U?" is given. This error message does not discard all of your type-in, just the undefined symbol and the following operator. If you wish to finish the command you started, you should continue with the correct spelling of the symbol and go on from there. Alternatively, if you

wish to make a forward reference to the undefined symbol, you can type just "?"; you need not retype the symbol's name. Of course, if the command so far is a complete mistake, you can use ^D to cancel all of it.

In its attempt to print PDP-10 instructions in a form intelligible to the user, DDT tries to find symbolic representations for addresses that it types out (see "Typeout Modes). But this introduces the problem of how to decide which symbol is appropriate. If the address 405 is to be typed out, and there are two symbols, START and FOOFLG, with value 400, either START+5 or FOOFLG+5 might be used. DDT can't tell which one is better, but if the author of the program knows that START is an address and FOOFLG the name of a bit, he might prefer to see START used. He can tell DDT never to use FOOFLG ever for symbolic typeout by "half-killing" it. Bit typeout mode is an exception; it is willing to use half-killed symbols, and must be, since bit names usually are half-killed. Half-killing is so useful that MIDAS provides commands to define and half-kill a symbol all at once ("::" and "=="); MIDAS communicates the halfdeadness of the symbol to DDT along with the symbol's value. In addition, there are DDT commands to half kill a symbol:

<symbol>◊K

half-kills the symbol <symbol>, so that it will not be used for typeout.

◊◊^C

half-kills the last symbol DDT typed out, and then tries again to print the last quantity printed. An example makes this clear: after DDT prints "MOVE A,FOO", if you decide that FOO is not the appropriate symbol, ◊◊^C will half-kill FOO and then try again to print the same MOVE instruction. This time it might come out as "MOVE A,BAR+3" if BAR is now the closest symbol to the address in the instruction.

<symbol>◊◊K

fully kills <symbol>. It is no longer defined (in the current job). Predefined symbols cannot be killed or half-killed. However, a definition in the current job's symbol table overrides any built-in dfefinition of the same symbol.

Symbols normally accompany programs. A binary file usually contains a symbol table, and when DDT loads a binary file into an inferior job it usually also remembers the symbol table from that file as the symbol table of the job. When DDT dumps the core of a job into a binary file, the job's symbol table is also written. At times this process does not do the right thing automatically, so commands are provided for manipulating symbol tables:

◊◊K

deletes the current job's entire symbol table. It is equivalent to fully killing each of the symbols with <symbol>◊◊K.

:SYMLOD <file> or :SL <file>

reads the symbol table from the binary file <file> and makes it the new symbol table of the current job. Any symbols the current job previously had are killed. This command is useful when examining a job that somehow (such as by being disowned and reowned) came to have no symbols. :SYMLOD uses the same filename defaults as the loading and dumping commands. :SYMADD is a similar command the keeps the symbols the job used to have in addition to the new ones.

◊^K

is the same as :SYMLOD<cr>, and loads the symbol table out of the ◊L

default file. It is useful after a file has been loaded without symbols and the symbols later appear to be necessary after all. The usual reason that a file was loaded without symbols is that it was loaded by a command to run a program, such as <prgm>^K or :<prgm>. Notice that <prgm>◊^K is equivalent to <prgm>^K with a ◊^K done before starting the program.

:LISTS

lists the names of all the symbols in the current job.

A program's symbol table may be arranged in a block structure which limits each symbol's scope. There may then be several symbols with the same name, defined in different blocks. Block structure will exist in the symbol table of a MIDAS program only if the program explicitly makes use of MIDAS block structure with the .BEGIN and .END pseudo-ops.

DDT handles block structured symbol tables by remembering, at all times, a currently selected symbol table block for each job. All references to symbols use the selected block as the scope. However, for ease of use, if a symbol is not, strictly speaking, accessible from the selected block, but it is defined in some other block, the other block's definition is still visible. Use of such a symbol as input will select the block that the symbol is defined in, leaving the previously selected block. The user will be notified with a message such as "<newblock>◊:". Not surprisingly, <newblock>◊: is a command that selects the specified block; see the reference section. :LISTP prints the block structure of the current job's symbol table, and :PRGM prints the name of the selected block.

At times it is necessary to store a copy of a job's symbol table into the job's memory, or to read symbol definitions or a whole symbol table into DDT out of the job's memory (the linking loader STINK does this to give DDT the symbol table constructed by the loading process). The commands ^Y, ◊^Y and ◊◊^Y serve those functions. The command :SYMTYP takes a symbol as argument and returns information on whether the symbol is defined. See the reference section.

Finally, some files do not contain any symbol table, but instead contain an "indirect symbol table pointer" to another file. This means that attempting to load the symbol table of the file containing the pointer will load the symbol table of the file pointed at. If you load a system program with ^K (no symbols), then dump it after running it, it will automatically be given an indirect symbol table pointer to the system program file that was loaded.
v

File: DDT, Node: Memory, Previous: Symbols, Up: Top, Next: Insns

Examining and Altering Memory

In debugging, the most important operations are examining the job's memory locations and depositing new data in them. In DDT, words are examined by commands to "open" them. Once a location is open, new contents can be deposited in it; it is impossible to deposit in a location unless it is open. At any time there may be at most one location open; opening one location automatically "closes" the previously open location. Many commands (including all commands that print a "*" when they are done) close any open location without opening another. This is to make sure that you do not accidentally deposit in a location which had remained open so long that that fact was no longer obvious.

Opening a location usually prints out the contents of the location. Just how the contents are printed is up to the user, who can select from several built-in "typeout modes" including symbolic mode, numeric mode, ascii text mode, etc. (See the section on typeout modes).

The simplest and most frequently used way to open a location is the "/" command. It is preceded by the address of the location to open. The contents of the location are typed out, using whatever typeout mode is currently selected. In addition, the address and the contents are remembered as the values of the special symbols "." and ◊Q, respectively. Here is an example, which assumes that symbolic typeout mode is selected (as is usually the case):

```
        107/   MOVE A,FOO+3
```

"107/" was typed by the user, and the contents of 107 were printed, as a PDP-10 instruction, by DDT. After this command, the value of "." will be 107, and the value of ◊Q will be MOVE A,FOO+3.

If you try to examine a location at which the current job has no memory, the error message "??" will be printed. No location will be open.

Having opened location 107 and seen what is stored there, you can now deposit new contents with the <cr> command.

<newstuff><cr>

closes the open location, after depositing <newstuff> in it. If no location is open, this command has no effect, except that in either case ◊Q is set to <newstuff>. If you deposit in a read-only page, DDT replaces that page of the inferior's memory with a new, writable page, containing the same data in each word, before depositing. This can cause a page which was once shared with other jobs to become private, which occasionally causes a problem, so DDT informs you with a message like ":UNPURE <address>" (ref :UNPURE, ◊◊^M, ..UNPURE).

<cr>

with no argument simply closes any open location, preventing inadvertant modification of its contents. Other DDT commands that do depositing all do it just like <cr>; each deposits its argument in the open location, if there is one, as the very first thing it does. They differ from <cr> in what they do after depositing. For example, some go on to open other locations.

<cr>, with or without arguments, has another important effect: it undoes any temporary typeout mode selections, reverting to the permamently selected typeout modes. The significance of this will become clear in the section on typeout modes. Only <cr> performs this function. The other commands that deposit do not do so.

Often when one location is interesting, the one after it is also interesting. The command <lf> opens the location following the last opened location (that is, the one whose address is 1 larger). It moves to a new line and prints the address of the location it is opening, followed by a slash, before it prints the contents. For example, if after opening 107 as above you type <lf>, the typescript might appear as follows:

```
107/   MOVE A,FOO+3
START+10/   JRST START1
```

This assumes that START has the value 100 (a likely possibility), so that START+10 is just the symbolic expression of 110. The entire line starting with START has been printed by DDT, but since your type-in can't be distinguished from DDT's output, you could have typed in <cr> and then START+10/ and produced an identical script. That's a feature, not a bug, though: <lf> is defined to do exactly START+10/, so it might as well look the part. Commands which open a location whose address is not immediately visible as an argument in the command often print the address just as <lf> does.

<newstuff><lf>

is equivalent to <newstuff><cr><lf>. If there is an open location, <newstuff> is deposited in it and the next location is opened. If there is no longer an open location, <lf> moves one word down from the last location to have been open, but in this case <newstuff> is not deposited.

To "undo" a <lf> command, use the ^ command, which opens the word before the last word opened, instead of after. Doing <lf>^ when location 107 is open will open first 110 and then 107 again. ^ with an argument can be used for depositing, just like <lf>.

After seeing the instruction MOVE A,FOO+3 typed out, you might wonder what is in FOO+3. DDT saves you the trouble of typing FOO+3 in again by providing the <tab> (or ^I) command. <tab> opens the location addressed by the right half of ◊Q, which, after the 107/, would be FOO+3. <tab> prints the address it is opening on a new line, just like <lf>. Alternatively, you can use the command / with no argument, which will also open the location addressed by the right half of ◊Q, but will not print that address or go to a new line. Giving an argument to <tab> will deposit that argument (if there is still a location open) and then open the location pointed to by the argument; this is very different from what / does with an argument.

More hairy examination commands exist, which either specify a typeout mode for printing the contents of the location, or obtain the address to open in an unusual way. The first class of commands includes], which prints the opened word's contents symbolically no matter which mode is current, and [, which always prints the contents numerically. The second class actually consists of one- and two-altmode variants of the commands already described: /,[,] and <tab>. Those commands all open a location whose address is taken from either an argument or ◊Q. With no altmodes, they get the address from the right half of the argument or ◊Q. The one-altmode variants, such as ◊/ and ◊<tab>, all take the address from the left half of the argument or ◊Q. Even more useful, the two altmode variants such as ◊◊/ perform a PDP-10 effective address calculation on the argument or ◊Q, and open the addressed location. For example, suppose that A contains 5 and 1734 contains ADD T,FOO(A). Then after 1734/, ◊◊/ would open location FOO+5. @A◊◊/ would open location 5.

The commands to access the address ring buffer also examine and deposit memory locations, but in view of their special functions they are described in a later section (The Address and Value Ring Buffers).
v

File: DDT, Node: Insns, Previous: Memory, Up: Top, Next: Literals

PDP-10 Instruction Type-in

One thing that one often wishes to deposit in memory is a PDP-10 instruction. DDT allows them to be expressed almost as they would be in the MIDAS assembler, with some exceptions: <tab>s are NOT allowed in place of spaces, since <tab> is a DDT command in itself; literals too are not allowed. However, spaces, commas, parentheses and angle brackets have about the same meaning as in MIDAS. An instruction is made up of one or more "fields" separated by spaces or commas. Each field is a number, a symbol, or an arithmetic expression. The fields are combined to form a word in a way that depends on the instruction's "format", which is the pattern of spaces and commas around the fields. Each format has a fixed meaning in DDT, usually the same as the format's default meaning in MIDAS. The formats include:

<value>

by itself simply evaluates to <value>.

,<rh>

truncates <rh> to 18. bits, and returns it as the right half of the instruction, with zero as the left half.

,,<rh>

acts like ,<rh>. ,,<rh> and ,<rh> both exist because they are special cases of two different formats.

<lh>,,<rh>

returns a word with <lh> in its left half and <rh> in its right half.

<lh>,,

returns a word with <lh> in its left half and zero in its right half.

<opcode> <addr>

returns a word with <opcode> added to <addr> (which is truncated to 18. bits). If <opcode>'s right half is zero, this puts <addr> by itself in the right half (address field) of the result. The most common use has a PDP-10 instruction name as the <opcode>.

<ac>,

returns a word with <ac> in the accumulator field, and zero elsewhere. This format in MIDAS normally does something else.

<ac>,<addr>

returns a word with <ac> in the accumulator field, and <addr> (truncated to 18. bits) in the right half). This format in MIDAS normally does something else.

<opcode> <ac>,

adds <ac> into the accumulator field of <opcode>.

<opcode> <ac>,<addr>

adds <ac> into the accumulator field, and <addr> into the right half (address field).

Notice the two formats, "<ac>," and "<ac>,<addr>", whose meanings

in DDT are not the same as their default meanings in MIDAS. Many people redefine those formats in MIDAS to mean the same thing they mean in DDT, and perhaps someday MIDAS will be changed to be compatible.

Just as in MIDAS, @ can be included in an instruction to set the indirect bit, and a parenthesized expression can be used to specify the index field. It makes no difference where in the instruction the @ goes, although it cannot be put in the middle of a number, symbol, or operator without causing syntactic trouble. A parenthesized expression's meaning depends on whether it follows directly an arithmetic operator. If it does, it acts like a term in the expression, whose value is the halves-swap of the expression inside the parentheses. If there is no arithmetic operator in front of the "(", then the parenthesized expression, like an @, has a global effect rather than a local one: the expression inside has its halves swapped and the result is added into the entire word, after the fields are merged according to the format. Thus, 2*(1) is 2*1000000, or <2,,>, and if it appears as the address of an instruction, it is truncated to 18. bits, giving 0. But 2(1) appears, in its context, as just 2, and the entire instruction has 1,, added to it putting a 1 in the instruction field. Since the 1 in the (1) bypasses the format-processor, truncation of addresses to 18. bits has no effect on it.

◊>

allows you to type in a new instruction slightly different from another one without typing in all the fields that are the same. ◊> included anywhere in a PDP-10 instruction causes all fields not specified to be taken over from the value of ◊Q. DDT's heuristics for deciding what fields to default are complicated (ref ◊>). Here are some examples, which assume that you or DDT just typed MOVE A,B(C):

Typing	Gives
FOO◊>	MOVE A,FOO(C)
FOO(0)◊>	MOVE A,FOO
MOVNS◊>	MOVNS A,B(C)
MOVNS 0,◊>	MOVNS B(C)
@◊>	MOVE A,@B(C)
0,,◊>	B
,0◊>	MOVE A,(C)

v

File: DDT, Node: Literals, Previous: Insns, Up: Top, Next: Text

Literals

Literals are just as useful when typing in code with DDT as they are in an assembler. However, because DDT can't assemble a word of code without knowing where it is to be deposited, literals in DDT don't work quite the way they would in an assembler.

DDT stores literals in the job's patch area. The symbol PATCH points to the next word available for use by a literal; as words are used for literals PATCH is updated. See the section on patching for how to create a patch area. Every program ought to have one.

◊◊(

requests a literal. Type this command in the expression from which you wish to refer to the address of a literal. In an assembler, you would follow it with the data in the literal. DDT, however, will ask you for the data when DDT can digest it. What it will do immediately is print out a symbol, such as "$LT001", and a close-parenthesis. You should then finish typing the expression and deposit it, knowing that the symbol DDT typed will be a forward-reference to the address of the literal.

But when do you supply the data for the literal? Usually, as soon as you deposit the expression that contained it, DDT will ask you to do so. But if you use nested literals, or use a literal while making a patch (see PATCHING, below), DDT will have to wait for a while before asking for the data. In any case, DDT will ask for the data eventually, by typing out "$LT001/ 0 ". The symbol name typed tells you which literal DDT is asking for - just match it up with the symbol printed by the ◊◊(before. At this time, a location in the patch area is open, and you should start by depositing the first word of the literal. Deposit as many words as you like, or do other things. When you are finished, open the word AFTER the end of the literal and type ◊◊). This tells DDT where it should start the next literal or patch.

<arg>◊◊)

deposits <arg> in the open location and tells DDT that it is the last word of the literal (so the next literal will start in the next word).

If you rub out a ◊◊(, DDT will still think that the literal needs to be defined and will eventually ask you to define it. But when that happens, you need only type ◊◊) immediately. The literal will have been defined to be zero words long, and DDT will be satisfied.
v

File: DDT, Node: Text, Previous: Literals, Up: Top, Next: Modes

Text Type-in

When altering text strings, you can use DDT's commands that compute the numerical representation of text according to the conventions used most often by PDP-10 programs: ASCII, SIXBIT and SQUOZE.

◊1'<text>◊

has, as its value, the SIXBIT representation of <text>. In other words, it is the DDT equivalent of SIXBIT /<text>/. Only the first 6 characters of <text> are meaningful. The "1" in ◊1' can be replaced by any other number without effect; it is present to distinguish this command from the ◊' command, which selects a typeout mode.

◊2"<text>◊

has, as its value, the ASCII representation of <text>. In other words, it is the DDT equivalent of ASCII /<text>/. Only the first 5 characters of <text> are meaningful. Special DDT characters such as ^V, ^D, ◊ and <rubout> can't be entered directly, so a quoting mechanism is provided. The character ^ "controlifies" the following character, so that ^ and V make a ^V. ^ and ? make a ^?, which is a <rubout>. ◊ is actually a ^[, so ^ and [can be used to enter it. In addition, ^Q quotes some characters such as <rubout>, ^, and ◊. To play safe, use quoting to input any control character.

The "2" in ◊2" has two effects: it distinguishes this command from the ◊" command, which selects a typeout mode, and its low bit specifies the low bit of the value. Thus, 0 could be used instead of 2 with no effect, but 2 is easier to type.

◊1#<char>

returns the ASCII representation of <char>, right-justified in a word. It is the equivalent of MIDAS "<char> or "<char>". ^Q and ^ must be used for quoting, as in ◊2".

◊<flags>&<symbol>

is the equivalent of SQUOZE <flags>,<symbol> in MIDAS. This construct does not contain any final delimiter; instead, the following operator serves as one.
v

File: DDT, Node: Modes, Previous: Text, Up: Top, Next: Type-out

Type-out Modes

The contents of a memory location may be interpreted by a program in many different ways. DDT knows several of the most common ways, in that it can print the value of a word by showing what it would mean given a desired method of interpretation. For example, "symbolic typeout mode" prints a word as a symbolic address or as a PDP-10 instruction containing one; "ASCII typeout mode" prints a word as five ASCII characters; "Constant typeout mode" prints a word as a number. Some typeout modes have sub-options. For example, those that print numbers will use whatever output radix has been specified.

At any time, one typeout mode is "selected" or "current". Most DDT commands that print the value of a numerical quantity will use the current typeout mode (some commands, that know the significance of what they are printing, always use the appropriate mode regardless of what mode is selected). DDT commands exist for selecting various typeout modes either temporarily or permanently. If a new mode is selected permanently, it remains selected until explicitly replaced. If the current mode is changed temporarily, the new selection remains in effect only until the next <cr> command (<cr>'s that are the argument-terminators of other commands, such as commands that read filenames, do not count in this regard). At that time, the last permanently selected mode will be reselected.

Each typeout mode that DDT has, has a command to select it. The commands to select a mode temporarily all have exactly one altmode. If a second altmode is used, the selection is made permanent. For example, ◊F selects typeout as floating-point numbers, temporarily. ◊◊F selects the same mode permanently.

When a typeout mode has sub-options (such as whether to print component addresses numerically or symbolically, or what bit-name prefix to use), those sub-options may also be set either temporarily or permanently. When the sub-option is set by an argument in the command that selects the mode, the sub-option is set temporarily if the mode is being selected temporarily; permanently, if the selection is permanent.

In addition to the temporarily and permanently selected modes (also known as the "temporary mode" and the "permanent mode"), DDT remembers which mode was most recently explicitly specified. This variable is updated just as the temporary mode is, except that it is NOT reset by carriage returns. The most recent mode can be used to type one value with the ";" command, or can be selected temporarily or permanently with "◊;" or "◊◊;". Thus, after temporarily selecting halfword mode with ◊H and then reverting to the permanent mode with <cr>, ";" will still type its argument in halfword mode.

The user may define typeout modes of his own, by supplying DDT with an instruction which, when executed in the inferior itself, will print a value in his favorite manner. DDT has several typeout-mode selecting commands that select a variable mode instead of a fixed one; if the user's instruction is specified as the definition of one of those modes, the user's instruction will be used for typeout when that mode is selected. The commands that select a variable mode are ◊", ◊#, ◊$, ◊%, ◊&, and ◊'. The terms "# mode", "$ mode", etc. mean "the mode that ◊# currently specifies", etc. Some of the variable modes are initially set to useful built-in typeout modes such as ASCII mode, SQUOZE mode, etc. Others (◊$ and ◊%) are useless unless given meaning by the user. The meanings of the variable mode commands are controlled by locations inside DDT: each job has one location for each of the six commands. The locations' addresses are called, respectively, ..TDQUOT, ..TNMSGN, ..TDOLLAR, ..TPERCE, ..TAMPER and ..TPRIME. Each variable can contain either -1,,<addr in DDT> or an instruction to be executed in the inferior job. No addresses in DDT

should be used except those designed for such use, which have names starting with ..TM: ..TMSQ is the address of the SQUOZE typeout mode, for example (ref ..TMSQ). The only user instructions that are likely to be useful are subroutine calls (including user UUOs); the subroutine should expect to find the value to be typed out in location 37 (but ref .40ADDR). In addition, the address of the open location will be in 25 (but ref .40ADDR), but depending on this makes the typeout mode less versatile (it can't be used in a Raid register, for example).

With the exception of ◊T mode, all of the built-in DDT typeout modes arrange for their output to be in a form suitable for being typed back in. That is automatic for ◊C, ◊H, ◊F and ◊S modes. For the ASCII, SQUOZE, and SIXBIT typeout modes, it requires that the data typed out be preceded by an appropriate DDT operator for reading the data back in, and that it be printed in the syntax used by that operator.

Here follow descriptions of the fixed built-in typeout modes.

◊C selects Constant mode, in which words are typed out as numbers, using the currently selected output radix.

◊E selects E&S mode, in which words are typed out as E&S display processor instructions.

◊F selects Floating point mode, in which words are typed out as floating point numbers. The radix is always decimal.

◊H selects Halfword mode, in which words are typed out as <lh>,,<rh>, with each halfword printed as an address.

◊S selects Symbolic mode, in which words are typed out as cleverly as DDT can manage, either as a PDP-10 instruction, in halfword mode, or as a number. In the first case, the address, index and AC fields are printed as addresses. In the other two cases, the ordinary ◊C and ◊H mode actions are used. The decision of which format to use is complicated, but PDP-10 instruction printout is used whenever it makes sense. With I/O instructions, the user can control the decision; see ..D010 in the section on Specially used symbols. The .CALL UUO is printed specially; DDT prints in parentheses the name of the call (eg, OPEN).

◊<n>T selects typeout of words as decomposed into <n>-bit bytes.
◊<pat>T selects typeout of words as decomposed into bytes in an arbitrary pattern specified by <pat>. The byte boundaries occur where two adjacent bits in <pat> differ. Thus, the pattern 707070,,631463 divides the LH into 3-bit bytes and the RH into 2-bit bytes.

◊T selects typeout of words as decomposed into bytes according to the last byte size or pattern specified.

Here follow the descriptions of the built-in typeout modes that are the initial settings of the variable commands ◊", etc.

◊" is initially set to select full-word ASCII typeout mode, in which ASCII /<foo>/ types out as ◊0"<foo>◊, and ASCII /<foo>/+1 types out as ◊1"<foo>◊. Control characters in <foo> are typed as uparrow followed by the appropriate non-control character. Uparrow is preceded by a ^Q. Rubout is typed as uparrow-?. Altmode is typed as uparrow-[.

◊# is initially set to select single-character ASCII typeout mode, in which the ASCII code for <char> is typed out as ◊1#<char>. As in full-word ASCII typeout mode, control characters are typed out with

uparrows and ^ and ^Q are preceded by ^Q.
Altmode, however, is typed as an altmode.

◊& is initially set to select SQUOZE typeout mode, in which SQUOZE <flags>,<symbol> types out as ◊<flags>&<symbol>. <flags> will always be a multiple of 4, and less than 100 .

◊' is initially set to select SIXBIT typeout mode, in which SIXBIT /<foo>/ is typed out as ◊0'<foo>◊.

Here are the commands that control how addresses are printed:

◊A selects typeout of addresses as numbers (in the selected radix).

◊R selects typeout of addresses symbolically, when possible. An address which is not equal to any symbol will be typed as <symbol>+<number>, provided that <number> is less than the current contents of ..SYMOFS (initially 100).

◊? selects bit typeout mode, which is complicated, and is described in a later section.

Here are the commands that select the output radix, in which numbers are printed:

◊D selects base 10.

◊O selects base 8.

◊<n>R selects base <n>.
v

File: DDT, Node: Type-out, Previous: Modes, Up: Top, Next: Bit

Type-out Commands

Several DDT commands print a numeric argument back out using a specific typeout mode. They exist for convenience, since it would be possible anyway to select that typeout mode and then use the ; command to print the argument in that mode. When one of these commands has no argument, it prints the value of ◊Q.

_ (underscore or backarrow)

prints ◊Q (or its argument, if there is one) symbolically.

=

prints ◊Q or its argument as a fixed-point number. ◊= prints as a floating point number.

◊<r>=

prints ◊Q or a prefix argument as a fixed point number using <r> as the radix.

'

prints ◊Q or its argument in the ◊' mode, which is SIXBIT text mode unless the user has changed it.

"

prints ◊Q or its argument in the ◊" mode, which is ASCII text mode unless the user has changed it.

#

prints ◊Q in the ◊# mode, which is single-character ASCII mode unless the user has changed it. # cannot be given an argument to print, since then # would be interpreted as an arithmetic operator.

&

prints ◊Q in the ◊& mode, which is SQUOZE text mode unless the user has changed it. & cannot be given an argument to print, since then & would be interpreted as an arithmetic operator.

?

prints ◊Q in ◊H◊? mode, which is a case of bit typeout mode (to be described later).

File: DDT, Node: Bit, Previous: Typeout, Up: Top, Next: Pseudo

Bit Typeout Mode

Bit typeout mode makes it possible to interpret a word in terms of particular sets of related symbols. For example, a word can be decomposed into a sum of several symbols that are the names of flag-bits in a particular location. The restriction is that all of the symbols' names must start with the same prefix, since that prefix is how DDT is told which symbols to use. When defining a set of bit names in a program, it is wise to make them start with a common prefix for the sake of bit typeout mode.

In addition to the prefix, bit typeout mode requires a byte decomposition pattern, such as the ◊T typeout mode uses. This tells DDT how to divide the quantity being typed into bytes. The symbols typed are not allowed to overlap the byte boundaries, and each one must completely account for the value of one of the bytes. This restriction usually prevents any trouble from unrelated symbols that happen to begin with the specified prefix.

The convention ITS uses for naming flag bits gives each flag word two prefixes, one for LH bits and one for RH bits. Therefore, bit typeout mode is actually applied to one halfword at a time. Each bit typeout prefix can specify that it applies only to a single halfword. In addition, it is possible to have two different bit typeout prefixes selected in DDT at a time, one for RH bits and one for LH bits. The mechanism is this: in DDT, there is a "main selected bit typeout mode" variable, and an "alternate selected bit typeout mode" variable. Each one can contain a prefix and a byte decomposition pattern. When bit typeout mode itself is enabled, the main bit typeout mode is used for whatever halfwords it applies to; when it does not apply, the alternate bit typeout mode is used if it applies.

When a new bit typeout prefix is selected, it normally becomes the main selected bit typeout mode. The previous main selected mode becomes the alternate.

The main and alternate bit-typeout prefixes are in ..BITS and ..BITS+1 as SQUOZE values.

The main and alternate byte-decomposition patterns are in ..BITP and ..BITP+1.

Assume from now on that the prefix is "%TX". It in fact may be any number of characters long, and is a prefix for bit names.

When %TX bit-typeout mode is set, the byte-decomposition mask is determined. This is the value of the symbol "%TX", if it is defined; otherwise, the value of "..B%TX", if that is defined; otherwise 525252,,525252 octal. (The byte-decomposition mask may also be set explicitly by specifying it as an infix argument to ◊? or ◊◊?.) The byte-decomposition mask divides the word into fields in much the same manner as the ◊T mask does. If the byte-decomposition mask is negative, then it divides the word into fields. If it is positive, then its right half is divided into fields, and bit 3.1 determines the half (0 = RH, 1 = LH) which the mask applies to.

Bit-typeout mode is actually superimposed on other modes:

> ◊H typeout types the left half using left-half bits, and the right half using right-half bits.
>
> ◊S typeout, if it converts itself to ◊H, follows ◊H rules. Otherwise, the instruction is typed as usual, except for the address field, which may or may not be typed as as bits, according to the type of instruction. For example, TLNE uses

left-half bits, TRNE uses right-half bits,
HRLI uses left-half, HRRI uses right half,
HRRZ does not use bits, JRST does not use bits.
These op-codes use left-half bits:

```
        MOVSI
        HRLI    HRLZI   HRLOI   HRLEI
        TL--
```

These op-codes use right-half bits:

```
        MOVEI
        SETCMI  SETMI
        HRRI    HRRZI   HRROI   HRREI
        ANDI    ANDCAI  ANDCMI  ANDCBI
        IORI    ORCAI   ORCMI   ORCBI
        XORI    EQVI
        TR--
```

Op-codes which do not use bits always use the most
recent setting by ◊A or ◊R. "?", which means "◊H;",
can always be used to see bits explicitly
as left-half,,right-half, if ◊S doesn't give
exactly what is desired. To see "the other kind"
of bits, ◊◊Q? can always be used.

mode uses ◊S mode after extracting the low seven
bits, and so follows ◊S rules.

These modes use bit-typeout iff the bit-typeout flag is set.

The bit-typeout algorithm proceeds as follows: for each field of the byte-decomposition mask, examined in order from left to right, which contains nonzero in the value being printed, use that field to mask the quantity to be typed out. Look up this value in the symbol table. If a symbol starting with %TX is found with that value, print it. Otherwise, look up the value consisting of a 1 right-justified in the field; if a symbol beginning with %TX is found, type out <n>*%TXFOO where <n> is the value in the field, typed as a number in the current output radix, and %TXFOO is the symbol found. Otherwise, look up a mask for the whole field; if a symbol beginning with %TX with that value is found, type out <n>&%TXFOO, where <n> is the number being printed in bit mode, AND'ed with the field being handled. In this case, <n> and <n>&%TXFOO have the same value. The &%TXFOO is there to indicate what the <n> means. If none of those alternatives is successful, the field can't be typed with bit typeout mode. After trying to use bit typeout mode on each of the nonzero fields, those that failed, if any, are typed as a single address, absolutely or relatively according to the current typeout mode.

If left-half bits from a full-word byte-decomposition mask are being used to print out a half-word, the names of the bits are enclosed in parentheses. Thus:

```
        TLNE TT,(%QXABC+%QXDEF+%QXGHI)
```

Example:

```
        201140000701    %TX◊?#
```

would type out

```
        MOVEI C,%TXMTA+%TXCTL ◊1#A
```

or something like that.

Example: consider these definitions in a program:

```
%QXSYM==400000          ;funny bits
%QXLET==200000
%QXNUM==100000
                        ;some bits not defined
%QXCNT==7000            ;a field
$QXERS==700             ;another field
%QXERS==100             ;name for its low bit
$QXQTY==77              ;another field
%QXQTY==1               ;name for its low bit
```

Then these quantities would type out as follows:

•

```
Quantity        Bit Typeout

43              43#%QXQTY
123456          %QXNUM+3000&%QXCNT+4#%QXERS+56#%QXQTY+20000
500000          %QXSYM+%QXNUM
665400          %QXSYM+%QXLET+5000&%QXCNT+4#%QXERS+60000
```

There are predefined bit typeout prefixes for all the the series of system symbols starting with "%"; for example, "%TS" for the symbols %TSFRE, etc., for the bits in TTYSTS variables. In addition, there are the prefixes .R and .S which make it easy to find out which variable a .SUSET or .USET is reading or setting. Prefixes ..R and ..S serve the same function for .BREAK 12,'s.

v

File: DDT, Node: Pseudo, Previous: Bit, Up: Top, Next: Rings

Pseudo-locations: DDT Variables and ITS User Variables

Besides its memory, a job has many other variables which contain part of its status. Both DDT and ITS keep information about the job. DDT makes this information accessible to the user by allowing "pseudo-locations" that appear to contain the information, and which can be examined and deposited in as if they were part of the job's actual address space. Each frequently useful variable has its own symbol, whose value is the pseudo-location containing that variable. For example, .PIRQC is defined to be the pseudo-address of the current job's interrupt request word, which can be examined with .PIRQC/ and altered with <newstuff><cr>.

These "funny symbols" are the only symbols whose values are not precisely like numbers; the value of a funny symbol includes both a number and a flag indicating whether the value is an ITS variable or a DDT variable. In the case of a DDT variable, the numeric part of the value is the address in DDT where the information is actually stored. Some of the DDT pseudo-locations that have predefined symbols contain information associated with a single job, while others apply to all jobs or have nothing to do with specific jobs. The symbols for job-specific pseudo-locations have different values (point at different words in DDT) depending on which job is current. Funny symbols can't be defined by the user; the predefined ones are all there are. A complete list is in the reference section "Specially Used Symbols".

Another command that provides information about a job useful in debugging is :ERR, which can be used to decode the last system call error received by the job, or to tell the meaning of a specific system call error code. See the reference section.
v

File: DDT, Node: Rings, Previous: Pseudo, Up:Top, Next: Execution

The Address and Value Ring Buffers

DDT remembers several of the quantities most recently read or printed. The user can access those saved quantities without going to the trouble of typing them in full. The remembered quantities are stored in two "ring buffers", one for addresses opened, and one for values found by examining, printed out, or deposited into memory. Addresses or values that would otherwise be thrown away are pushed onto the front of a ring buffer, where they remain until squeezed out the back by later arrivals. Thus, the ring buffers always contain a record of a certain amount of recent history, ordered latest frontmost. The saved history can then be accessed by commands specially provided for that purpose.

Every value found by examining memory, deposited into memory, or printed out by a "retype using ... mode" command such as "=", ";", or "_", is pushed onto the value ring buffer. The contents of the ring buffer are accessible via the various forms of ◊Q:

◊Q

evaluates to whatever is at the front of the value ring buffer. It stands for the last thing DDT read or printed.

◊<n>Q

is the <n>'th value back in the value ring buffer. ◊0Q is the same as ◊Q. ◊1Q is the thing DDT read or printed before it read or printed ◊Q.

◊◊<n>Q

is ◊<n>Q with its halves swapped: <(◊<n>Q)>.

One use for ◊<n>Q is to move a series of words up or down one word. That is done by depositing ◊2Q into each word:

```
103/   51   0
104/   57   ◊2Q
105/   3    ◊2Q
```

When depositing into 104, ◊Q would be 57, ◊1Q would be 0, and ◊2Q is 51. The 104 does not count because it is used as an address, and will go on the address ring buffer instead of the value ring buffer.

The address ring buffer works in a more complicated way, because of heuristics designed to maximize its usefulness. When a location is opened, it is usually pushed onto the address ring buffer, but there are some exceptions. For one, if the address is the same as the one already at the front of the ring buffer, it is not pushed a second time. For another, the <lf> and ^ commands do not push a new address; they just replace the address at the front of the ring buffer with the new address, 1 larger or 1 smaller. These actions are designed to make the ring buffer remember history at a slightly higher level, ignoring small changes to have room for more big ones.

The address at the front of the address ring buffer is always the value of the special symbol ".". Thus, "./" will reopen the last word opened, and ".+2/" will open the second word down from it. Aside from that, the address ring buffer is accessed destructively, by discarding recent addresses from the front to get at the earlier addresses behind them.

◊<cr>

discards the address at the front of the address ring buffer, and

reopens the address which thereby appears at the front. After 1/ and 2/, ◊<cr> will discard the 2 and reopen 1. The reopened address is typed out symbolically on a new line, followed by slash and the contents of the location (in the current mode). ◊<n><cr> is an extension of the ◊<cr> command; it discards the first <n> addresses from the buffer and then reopens the one left at the front. When ◊<cr> (or the following commands, ◊<lf> and ◊^) is given a prefix argument, that argument is deposited into the open location (if there is one). In that regard, ◊<cr> is just like <cr>. However, ◊<cr>, unlike <cr>, does NOT revert to the permanently selected typeout modes.

◊<lf>

is like ◊<cr>, but instead of reopening the location that comes to the front of the ring buffer, it opens the word after that location. ◊<lf> is like an ◊<cr> followed by a <lf>, except that only the second location - the incremented address - is actually opened. When you have been examining a sequence of words with <lf>, and then digress to a word out of the sequence (with <tab>, for example), ◊<lf> will open the next word of the sequence. ◊<n><lf> is also defined, and pops <n> addresses off the ring buffer.

◊^

is like ◊<lf>, but decrements the address instead of incrementing it. ◊^ is a combination of ◊<cr> and ^. ◊<n>^ is also defined.
v

File: DDT, Node: Execution, Previous: Rings, Up: Top, Next: MAR

Controlling Execution While Debugging

This is an overview of the commands useful for controlling the execution of a job being debugged.

The basic commands for controlling execution are still important: ◊G to start the job at the program's start address, ^Z and ^X to stop it, and ◊P and ^P to resume execution.

DDT can impose on an inferior conditions under which it should stop executing. There are several types of conditions available. Breakpoints stop the inferior if it tries to execute a specific instruction; the "MAR" stops the inferior if it tries to refer in any way to a specific location. The user can supply further restrictions on a breakpoint or the MAR - that is, cause the breakpoint or MAR condition to be ignored unless certain other requirements are met - but cannot alter the fundamental way in which the breakpoint or MAR is triggered. Each job has eight breakpoints, and only one MAR (a hardware limitation), each of which can be set or disabled. The breakpoints are numbered from 1 to 8. The MAR and breakpoints are described in detail in later sections.

Also, the user can make the job stop before each system call by putting zero in the pseudo-location ..SYSUUO. When the job stops for such a reason, DDT gives "SYSUUO;" as the reason. ◊P will make it proceed on, but the next system call will make it stop again. Similarly, putting 0 in ..PERMIT will make the job stop before all .VALUEs, suicide attempts (.BREAK 16,), and .BREAK 12,'s that would write in DDT, with "DDTWRITE;" as the reason.

"Stepping" is running a job "slowly", so that its actions can be observed. DDT has commands to step either one instruction or one subroutine call, and to run through a program by repeated stepping. They are described in the section "Stepping".

Some other execution-control commands are these:

<addr>◊G

starts the program at address <addr>. The "first part done" hardware flag (/PCFPD) is cleared.

<addr>◊◊G

starts the program at address <addr>, and makes <addr> the new starting address for future ◊G's with no argument.

<addr>◊0G

sets the program's PC to <addr>, but doesn't start execution. <addr>◊0G followed by ◊P is equivalent to <addr>◊G. ◊0G sets the PC to the starting address.

<insn>◊X

makes the current job execute the instruction <insn>. If <insn> is a subroutine call, the whole subroutine is executed. If <insn> is a jump, the job will continue running until stopped for some unrelated reason. If the instruction returns without skipping, DDT prints one blank line and then a "*". If the instruction skips, DDT prints two blank lines before the "*". In either case, the job's PC is not altered by the ◊X.
v

File: DDT, Node: MAR, Previous: Execution, Up: Top, Next: Breakpoints

The MAR

For each inferior of DDT, there is one MAR, with which the user can make the inferior stop on referencing one particular word of memory. The MAR can be restricted to certain types of references, and an arbitrary conditional instruction can be supplied.

<addr>◊<mode>I

sets the current job's MAR to trap only some references to <addr>. The value of <mode> determines which types of reference are trapped: 1 traps instruction fetches; 2, write references; 3, all references (KL-10's allow a few other values that are less useful). <mode> can be omitted, in which case it defaults to 3 (all references).

When the MAR traps (or "is hit"), the job may be stopped either before or after executing the instruction, according to the mood of the hardware (on KL's, it is always before, which is useful; on KA's it is usually after but can sometimes be before). In either case, the job's PC will be "correct", so that it will not skip an instruction or execute one twice. But since the next instruction to be executed might not be the one which tripped the MAR, DDT in addition to that instruction prints the instruction which hit the MAR, and its address. Those two instructions might or might not be the same. When the MAR aborts the instruction that trips it, that instruction will be temporarily immune to MAR when the job is restarted (otherwise, it would be impossible to pass by that instruction).

◊I

turns off the current job's MAR. Reloading the job with a ^K command also turns it off (but ◊L does not!).

An additional condition can be imposed on the MAR by depositing an instruction in ..MARCON. When the MAR is tripped, DDT will put that instruction in the job's memory and execute it; if it fails to skip, DDT will ignore this particular triggering of the MAR. One common thing to do is to test the sign of the word that a write-catching MAR is set on. ..MARCON is reset to 0 by ◊I commands to minimize confusion.

In addition, ..MARXCT allows you to specify DDT commands to be executed when the MAR is hit. ..MARXCT, if nonzero, should be the address in the job's memory of the ASCIZ string of commands. Like ..MARCON, ..MARXCT is zeroed by ◊I commands.

File: DDT, Node: Breakpoints, Previous: MAR, Up: Top, Next: Stepping

Breakpoints

As said above, DDT gives every job eight breakpoints, each of which allows the user to make the job stop if it tries to execute one particular instruction. The breakpoints are numbered 1 through 8, and each one can be set on an instruction or disabled.

<addr>◊B

sets a breakpoint on the instruction at address <addr>. This involves replacing that instruction with a special breakpoint system call. However, that replacement is in effect only while the job is running (because DDT carefully makes the switch when starting the job and unmakes it when stopping the job), so examining the location while the job is stopped will show no change. Because breakpoints work this way, they cannot safely be put on locations which are used as data (that is, referenced other than by executing them). Also, if the program overwrites the breakpointed location, the breakpoint will be rendered ineffective. If you suspect that one of these problems is screwing you, try using the MAR instead; it is immune to them.

<addr>◊B chooses the lowest-numbered breakpoint that is not already in use; if the breakpoints are all in use, it is an error. In that case, you can use :LISTB to find out what breakpoints are set, and then clear some of them, or simply change their settings with <addr>◊<n>B. There are several ways to clear breakpoints:

0◊<n>B

clears breakpoint number <n>.

<addr>◊0B

clears the lowest numbered breakpoint set at <addr>.

◊◊B

clears all breakpoints (of the current job).

◊B

when stopped at a breakpoint, clears that breakpoint.

When a job stops at a breakpoint (breakpoint 3, say) and returns to DDT, DDT's message to the user looks like

◊3B; <pc> >> <insn>

<pc> is the job's PC, and will usually be the address where the breakpoint was set (but might be the address of an XCT instruction pointing there, etc). The status of this job in a :LISTJ would now be "3B".

Unless DDT is told otherwise, it will stop the job whenever a breakpoint is hit. However, for each breakpoint, you can change that. Giving a breakpoint a "proceed count" will make it count a certain number of hits before stopping the job. In addition, the breakpoint can have a conditional instruction, which will be tested each time the breakpoint is hit, and can make DDT stop the job sooner than the proceed count would require.

These more esoteric breakpoint features are accessible by modifying the four-word block in DDT that controls the breakpoint. There is a special way to refer to the beginning of breakpoint <n>'s four-word block: ◊<n>B is its address. ◊<n>B+2 holds the proceed count, and ◊<n>B+1 holds the conditional instruction. Ref ◊<n>B. In addition,

after stopping at a breakpoint, <n>◊P will continue and give that breakpoint a proceed count of <n> - it restarts the program not just until the breakpoint is next hit, but until the <n>'th time it is hit (of course, other things can still stop the program).
v

File: DDT, Node: Stepping, Previous: Breakpoints, Up:Top, Next: Raid

Stepping

When a bug has been brought to bay, the simplest way to flush it out is to step through the program one instruction or a few instructions at a time, watching things happen. DDT has commands to make this easy to do. The Raid register feature (described in a later section) is especially useful while stepping.

The basic stepping command is ^N, which runs the current job for one instruction, and then stops it and prints the next instruction (the one that will be executed first if the job is started again). This is called "one-proceeding". One-proceeding through a jump still executes only the jump; the job returns to DDT with the PC set to the address jumped to. ^N uses a special hardware feature that interrupts the inferior as soon as an instruction is completed.

<n>^N

runs the current job for <n> instructions, then stops it. If <n> is omitted, 1 instruction is executed.

Often one of the instructions in a path will be call to a subroutine which is above suspicion. When that happens, one wishes to regard the entire subroutine call as a single instruction and step through it all at once.

◊^N

executes one instruction, regarding subroutine calls as black-box single instructions. ◊^N defines "one instruction has executed" as "the PC is 1 or 2 greater than it started out". This is the right way to step over a subroutine call provided that the subroutine will return to one of the two locations following the instruction that called it. ◊^N places two breakpoints of a special kind, called temporary breakpoints, on those locations, to stop the program when the subroutine returns. ◊^N attempts to understand recursive subroutine calling and not stop until the stack level returns to its previous level. Another use for ◊^N is, after checking out the body of a loop on one iteration, to let the remaining iterations happen: just do ◊^N when the next instruction to be executed is the conditional branch back to the start of the loop. Temporary breakpoints differ from ordinary breakpoints in that they are both removed if either one of them is hit, unlike ordinary breakpoints which remain until explicitly removed.

◊<nargs>^N

steps over a subroutine call which is followed by <nargs> in-line arguments. The temporary breakpoints are put in the two locations following those arguments. Note that ordinary ◊^N, with no infix argument, does not assume that there are no arguments, as implied above. It tries to figure out how many there are, assuming that words with op-code fields (top 9 bits) containing 0 or 774 to 777 are probably arguments.

Another useful operation is to run a program to a certain point. For this, temporary breakpoints are just the thing:

<addr>◊^N

runs the current job until it reaches <addr>. This is done by putting two temporary breakpoints at <addr> and <addr>+1, and ◊P'ing. When control reaches <addr>, the temporary breakpoints will stop the job and be removed.

<stack pointer>,◊^N

runs the current job until it POPJ's. <stack pointer> should be an accumulator on which a PUSHJ was done; DDT finds the return address in the word on the top of the stack and puts temporary breakpoints there. Like ◊^N without arguments, this form of ◊^N tries to guess how many arguments followed the call, unless you tell it explicitly with an infix argument.

-1(<stack pointer>)◊^N

is the thing to use to return from a subroutine which has already pushed one word onto the stack - or to return from the second subroutine out. It works like <stack pointer>,◊^N except that the word one down on the stack, instead of the word at the very top, is assumed to hold the return address.

Lazy debuggers can tell DDT to step again and again, as if multiple ^N's or ◊^N's were being typed. This is called "multi-stepping" and is invoked with the command ^^. When DDT is multi-stepping, it prints one instruction and executes it, then prints the next instruction and executes it, until either the user types a command or a prespecified stop condition is met. There is a pause between the printing of an instruction and its execution, so that the user has a chance to see it and perhaps type a space to stop stepping. The available presettable stop-conditions include stopping before subroutine calls, stopping before system calls, stopping before jump instructions, and stopping before subroutine returns. In addition, multi-stepping can either step over subroutine calls instantly with a ◊^N, or step through the whole subroutine body with ^N's. The presettable options are called the stepping flags, and each job has its own settings of them. In addition, there are master settings used to initialize the flags of newly created jobs. The commands to alter the stepping flags are ◊^^ and ◊◊^^; see the reference section for details.

^^

begins multi-stepping, without altering the stepping flags.

<arg>^^

begins multi-stepping, but stops after <arg> steps.

0^^

prints the next instruction to be executed, but takes no steps. 0^^ is useful if you forget where your program stopped.

<addr>◊<n>G

sets the PC to <addr> and does <n> steps. This command is equivalent to <addr>◊0G <n>^^.
v

File: DDT, Node: Raid, Previous: Stepping, Up: Top, Next: Searches

Raid Registers

Raid registers are DDT's display feature, named after the display-oriented debugger at SAIL which suggested them. Every job has several Raid registers, each of which can be used to cause automatic display of the contents of one location in a specified typeout mode. Every time the job returns to DDT, all of the Raid registers that are set are displayed at the top of the screen. Each register is displayed both in the mode remembered in the Raid register, and as a constant in the remembered output radix. The current mode has no effect, and is not changed.

The main Raid-register control command is ◊V. Depending on the arguments it can set or clear a Raid register, or simply change the typeout mode associated with a register already set. In addition, ◊V always redisplays the Raid registers, even if it does nothing else. <addr>◊V sets a Raid register to display the contents of <addr>. Whatever typeout mode is current when the ◊V is done is remembered by the Raid register. If <addr> has the indirect bit or an index register in it, the address calculation will be done again each time the Raid register is to be displayed; this makes it easy to display, for example, the second word down on the stack (but you must use ",-1(P)", not simply "-1(P)". -1(P) equals -1+(P), which is NOT what you want. This screw is because, with no opcode preceding the -1, it is not truncated to 18. bits). <addr>◊0V clears a Raid register set on <addr>. ◊<n>V sets the typeout mode remembered by Raid register <n> to the current mode. ◊V by itself simply redisplays the Raid registers. Other options exist; see the reference section.

Raid registers can be used for dynamic examination of the rate of processing in a running program, by displaying the rate of change of the contents of a location (assumed to contain an integer). :RATE <addr> sets a Raid register to display the rate of change of <addr>'s contents, in terms of increments per millisecond. :ATB <addr> displays the average time between increments, or the inverse of the rate of change. When watching a running program, the :RAIDRP command is very useful. :RAIDRP <#sec> tells DDT to redisplay the raid registers every <#sec> seconds, stopping if you type any character.

Also relevant are :RAIDFL, which deallocates all of a job's Raid registers, and the block of storage starting at ..RAID in DDT, which contains three words whose contents direct DDT's actions (see the reference section).

Sometimes DDT will think that a raid register does not need to be redisplayed (its contents have not changed), when in fact it has been erased from the screen by other typing. When this happens, you should type ◊V with no arguments. This command is special in that it always redisplays all of the Raid registers, even those which have not changed.
v

File: DDT, Node: Searches, Previous: Raid, Up: Top, Next: Patch

Word Searches

There are DDT commands to find all words in the memory of a job whose contents meet a specified condition. The condition may be that certain bits do or do not all match a pattern, or that the word, regarded as a PDP-10 instruction, have a particular effective address (using the current contents, in the job, of any index registers and indirect address words required in the address calculation).

<value>◊W

finds all words in the current job which contain <value>. Each word found is opened as if by a <tab> command, printing the address and contents, and updating "." and the address ring buffer in the normal way. At any time, the search can be stopped with a ^D. ^D is synchronized with the printing process, so after a ^D the address ring buffer and "." will be set up as the printout would suggest. The range of core searched is specified by the contents of the ..LIMIT variable in DDT (ref ..LIMIT).

<low>◊,<high>◊,<value>◊W

finds all words between the addresses <low> and <high>, inclusive, that contain <value>. ..LIMIT is overridden.

<value>◊N

finds all words in the current job which do NOT contain <value>. In other respects just like <value>◊W. 0◊N is a convenient way to print all the memory of a job, saving lines by not mentioning words which are zero.

<mask>◊M

sets the mask for ◊W and ◊N searches. Only the bits of the word which are set in <mask> are considered by ◊W and ◊N when they compare a word's contents with <value>. The mask is initially -1 or 777777,,777777, so that all bits are compared. When set with an ◊M command, the mask keeps its new value until another ◊M command is done. Example: 777777◊M will make ◊W and ◊N compare only the right half; then 0◊N will find all words whose right halves are not zero.

There are actually eight different masks for word searches. Unless otherwise specified, mask number zero is used; that is what has been referred to as "the mask". But one can specify any of the other seven masks explicitly in an ◊M, ◊W or ◊N command with an infix arg: <mask>◊<n>M sets mask <n> and <value>◊<n>W searches using mask <n>. The other seven masks are useful mainly because they are initially set up to important subfields of a word. Mask 1 is set up to the right half; mask 2, to the left half. Masks 3, 4 and 5 are initially the AC field, index field and opcode. The eight masks are stored in a block in DDT starting at address ◊M, so that ◊M+3/ will examine mask 3 and allow you to change it.

<value>◊<mask>W

finds all words whose contents match <value> in all the bits specified by the mask. If <mask> is between 0 and 7, it is the number of one of the prespecified masks to use. Otherwise, <mask> itself is the mask to use. Thus, 5,,◊<-1,,>W compares all left halves against 5.

<address>◊E

finds all words in the current job whose effective address is <address>. Because ◊E must do an explicit PDP-10 effective address calculation on each word, it is much slower than ◊W and ◊N; usually,

◊1W (using mask 1 to compare just the right half) is a good substitute for ◊E. ◊E is not affected by the mask. It always compares all 18. bits of the effective address.
v

File: DDT, Node: Patch, Previous: Searches, Up: Top, Next: Services

The Patch Feature

The "patch feature" makes it easy to "insert" instructions into programs without reassembling them. In fact, what happens is that jump instructions are used to replace one instruction with several (possibly including a copy of the instruction replaced). Patches are inserted "carefully" so that there is no danger that a running program will try to execute a "half-made" patch and crash. Also, provision is automatically made for instructions that skip. The instructions in the patch are stored in the job's "patch area", a spare area allocated specifically to such use. Every program should allocate one. The beginning of the patch area is the value of PATCH if it is defined, or the value of PAT if it is defined, or 50 . As patches are made, PATCH will be redefined to point to the next free location in the patch area. If symbol table block structure is in use, PATCH must be in the global block to make sure that at any instant the same value of PATCH obtains in all blocks. To make sure of this, define it as global, even in an absolute assembly.

An ordinary patch is begun with the ^\ command, and ended (and made effective) with either ^] or ◊^]. The two main schemes of patches, patches "before" an existing instruction and patches "after" one, will now be described:

To patch "before" an existing instruction, open that location and type ^\. DDT will now open the patch area and reprint the instruction being patched over. Type a <rubout> to get rid of it. Then simply deposit the instructions to be inserted before the existing one, depositing the first instruction in the location opened by the ^\. After typing the last instruction, don't deposit it with <cr> or <lf>; instead, use ◊^]. ◊^] will deposit the new instruction, followed by a copy of the instruction being replaced, and the two JUMPA's back to the two locations after the one being patched. When that is done, the patch is finished. Here is an example: assume that 105 contains ADDI A,1 before which LSH A,1 must be inserted. Typing 105/^\<rubout>LSH A,1◊^] produces this printout:

```
105/   ADDI A,1   ^\
PATCH/   0   ADDI A,1<ADDI A,1> LSH A,1◊^]
PATCH+1/   0   ADDI A,1
PATCH+2/   0   JUMPA 1,106
PATCH+3/   0   JUMPA 1,107
105/   JUMPA 2,PATCH
```

To patch "after" an instruction, two things must be changed: The instruction must be deposited as the first of the patch, and it must not be put in at the end. Avoiding a copy at the end is done by using ^] instead of ◊^]. Putting a copy at the beginning is easy since DDT is already trying to supply one; just deposit it with <lf> instead of rubbing it out. Consider putting the LSH A,1 after the ADDI A,1 instead of before it. Typing 105/^\<lf>LSH A,1^] will do it, and produce this printout:

```
105/   ADDI A,1   ^\
PATCH/   0   ADDI A,1
PATCH+1/   0   LSH A,1^]
PATCH+2/   0   JUMPA 3,106
PATCH+3/   0   JUMPA 3,107
105/   ADDI A,1   JUMPA 2,PATCH
```

Notice that JUMPA 3, is used to return from the patch, instead of JUMPA 1,. JUMPA ignores its AC field, and the different numbers are used only as flags describing how to unmake the patch. The command ◊◊^\ exists to do that - see the reference section.

If you forget to follow the last instruction of the patch with ^]

or ◊^], and deposit it instead with <cr> or <lf>, you need not worry; just type the ^] or ◊^] and it will contrive to do the right thing. What is the right thing? After <insn><cr>, no location is open, and the right place for the first JUMPA is .+1. After <insn><lf>, the open location is the right place for the first JUMPA. So ^] with no argument, if there is a location open, stores the first JUMPA there; otherwise, it stores the first JUMPA in .+1. ◊^] acts similarly, except that it is the instruction being patched over, rather than the first JUMPA, that goes in the appropriate location. You can always count on being able to end a patch, no matter what has transpired, by opening the first word of the patch area whose contents should be clobbered by the return sequence, and then doing the ^] or ◊^].
v

File: DDT, Node: Services, Previous: Patch, Up: Top

DDT Services for Programs Running under DDT

DDT offers programs executing under DDT's control several services. There are defined conventions for normal termination, error reporting, reading or writing the DDT's information about the job, and passing DDT commands to execute.

Passing DDT a string of commands to execute is known as "valretting". It is done with the .VALUE instruction, actually a system call to cause a fatal interrupt which DDT responds to in a conventional way. The effective address of the .VALUE should point to an ASCIZ string containing the DDT commands. The conventions for those commands have been discussed above ("DDT Commands from Files"). Valretting is the most general way for a program to make use of DDT, and for that reason it is the ugliest way. Valretting can be done only by a job that has been given the control of the terminal; if a job which is running without the terminal tries to valret, the job is stopped and cannot actually valret until ◊P'd by the user. In addition, programs which valret are dependent on running under a DDT, since other superiors will probably be unable to make any sense of the valretted DDT commands. For these reasons, valretting should be used only when necessary. The ..PERMIT pseudo-location can be used to prevent a misbehaving program from valretting, or obtaining any service from DDT which might make DDT do unpredictable things. See the reference section.

A program can indicate normal termination to DDT with the .BREAK 16, instruction. The address field of the .BREAK 16, can be used to request various termination services from DDT. Unlike valretting, .BREAK 16, operations are understood to some degree by many of the programs that handle inferiors, and are thus safe to use in most programs. The address field is decoded bit by bit. Here is a list of what the bits mean. Bits that are good for general use are starred.

	bit	meaning if on
	2.9	◊X return, used by DDT to implement the ◊X command.
*	2.8	type an extra carriage return in DDT. Normally, .BREAK 16, causes DDT to type <cr><lf>*. With this bit, two <crlf>'s are printed. This is used to indicate that the program "did something" - it looks like what ◊X does when an instruction skips (can you guess why?).
*	2.7	do not reset teletype input (effective only if job has TTY) If this bit is not set, any typed-ahead input will be discarded, and any execute file or valret in progress will be suspended. This bit should always be used unless the program is reporting some sort of error.
*	2.6	kill this job, because it is finished. The job is killed immediately if it owns the terminal. Otherwise, it is killed when it is either selected with ◊J (with no arguments) or given control of the terminal. In any case, when the job is killed, DDT types ":KILL ", unless bit 2.5 or 2.3 is set.
*	2.5	kill this job as soon as possible. The job is killed instantly unless it is current but doesn't own the terminal. That exception is because it is very embarrassing for the current job to vanish while you are in the middle of typing a command. If the job is current and has the terminal, DDT prints ":KILL " as for bit 2.6. If the job is not current, it is killed immediately, with no notification to the user. 2.5 and 2.6 both set are slightly different from just bit 2.5: if the job is not current, it is killed instantly, but the user is informed with a "Job <jname> Finished" message.
	2.4	conditional breakpoint return, used by DDT to implement MAR and breakpoint conditional instructions. Bit 2.8, if on says that the condition is true.
*	2.3	inhibits all typeout associated with the .BREAK 16,. Bit

2.3 may be combined with the other options. It prevents the normal printout of <crlf>*; it prevents the printout of ":KILL " if the job is killed. In addition, if this bit is set, the open location is not closed. DDT uses bit 2.3 when invoking user-defined typeout modes, but its effects are simple enough to be described, so it is O.K. to use for other reasons.

If ..PERMIT is nonnegative, .BREAK 16,'s that kill the job or type nothing out are illegal, and DDT treats them like .BREAK 0,'s.

When a disowned job wishes to kill itself, it can do .LOGOUT. When an inferior wishes to kill itself, it can do .BREAK 16,160000 . If a program does not know whether it is disowned, and wants to kill itself in any case, it can do

.LOGOUT 1,

which acts like an ordinary .LOGOUT if the job is disowned, but interrupts the superior if there is one. DDT treats it just like .BREAK 16,160000 when it is executed by an inferior.

Often a program encounters an unexpected error in a system call and has no idea how to recover from it. In such a situation, the error should be reported to the user, so that he can eliminate the source of the trouble and tell the program to try again. ITS provides a system call LOSE and a UUO .LOSE as the conventional way to report such an error, and DDT responds to those instructions by describing the error to the user. Other superiors, such as batch job controllers, might want to try on their own to recover from the problem as reported. When .LOSE is suitable, it is preferred to alternatives which involve use of the terminal.

The .LOSE UUO is intended to follow an instruction which skips if there is no error. .LOSE backs up the PC to point once more at that instruction which failed to skip, and then causes a fatal interrupt which brings DDT onto the scene. Because the PC has been backed up, if the job is ◊P'd the problematical instruction will be executed once more, and if the obstruction has been removed in the meanwhile the program will proceed with its task. The most common instruction to put before a .LOSE is a system call:

```
        .OPEN CHN,[SIXBIT /   DSK/ ? SIXBIT /FOO/ ? SIXBIT />/]
         .LOSE %LSSYS            ;report failure of the .OPEN
```

the %LSSYS, or the address field of the .LOSE in general, says what kind of error happened, or where to find that information. %LSSYS in particular tells DDT to print an error message based on the last system-call error code. The allowed codes are described below.

For situations where .LOSE is too restrictive, the symbolic system call LOSE exists. Symbolic LOSE allows the new PC value to be specified explicitly, and therefore is suitable for use inside an error-handling routine. In addition, the address of the "culpable" instruction can be specified, although it defaults to the new PC value. Thus, the program can provide more complicated error recovery than simply restarting at the losing instruction. For details on symbolic LOSE, see .INFO.;ITS .CALLS.

The permissible values of the address field are subject to and given their meanings by a convention enforced by DDT's interpretation of them. They are:

1000 (symbol: %LSSYS)

is used to report that a system call unexpectedly failed, and the system's error code should be used to obtain the error message from the user. It may be used when a .OPEN or symbolic system call fails to skip, because those are the instructions that provide a system

error code to be decoded.

1000+<errcode>

reports an error and supplies a system error code <errcode> that describes it. DDT prints the standard error message associated with that error code. It is not necessary for the error detected by the program to have had that code; it need not even have had anything directly to do with a system call, since DDT uses only <errcode>.

1400 (symbol: %LSFIL)

is an improved version of code 1000, to be used in reporting system call errors that pertain to I/O and channels. DDT tries to find the name of the file on which the failing system call was operating, so it can tell the user. Code 1400 works with symbolic OPENs, and with system calls that use a previously opened channel, but not with .OPENs.

1400+<errcode>

is like 1400 except that instead of printing the error message associated with the job's last system call error code, it prints the message associated with code <errcode>.

1+.LZ <interrupt bit>

reports an error of the type described by the specified interrupt bit, to request DDT's normal handling of that particular interrupt. For example, 1+.LZ %PIMPV will make DDT tell the user that the job received a fatal MPV interrupt. Why might a program wish to do this? It might have enabled its own handling of MPV, and then received an MPV interrupt at a time when one was not expected and was not recoverable. At such a time the ideal thing to do is to report the MPV back to DDT, so that DDT will handle it - to "pretend" that MPV wasn't enabled at all. To make the pretense complete, the program's own MPV interrupt handler should dismiss the interrupt, and leave the PC pointing at the guilty instruction, since that would be the state of things if the program had not handled the interrupt. That can be done with a special feature of the DISMIS symbolic system call, which can do a .LOSE after dismissing the interrupt and restoring the PC. See ITS .CALLS for more details.

0

is a catch-all for errors that do not fall into the classes defined above.

The .VALUE instruction, with address 0, is the usual way to report an "impossible" occurrence. Unlike .LOSE 0, it leaves the PC pointing to the instruction after it, so the program can be ◊P'd with mimimum effort. Thus, .VALUE 0 is also useful for errors which are not very important, so the user might wish to proceed despite them. It is also used when the preceding instruction is of no particular relevance to the error, and there is no point in backing up the PC to it.

Inferiors may read and write various information from and into DDT using the .BREAK 12, instruction. The format of the arguments to .BREAK 12, is like that for .SUSET; the difference is that .SUSET is used for variables in ITS, and .BREAK 12, is used for variables in DDT. The .BREAK 12, should point at a word that has a sub-operation code in the left half, and the address of the area to be read or written in the right half. Alternatively, the word may be an AOBJN pointer to a block of words, each containing a sub-operation code and an address. The sub-operation code consists of a small number describing the type of information being read or written, and a read vs. write bit, which is 400000. Thus 400001 specifies writing the starting address. Writing with .BREAK 12, is not allowed if the job's

..PERMIT variable holds a positive number. The valid sub-operation codes have customary symbols defined in both DDT and MIDAS, starting with "..", followed by "S" or "R" indicating setting or reading, followed by three more characters mnemonic of the type of information. Here is a list of the defined sub-operation codes, followed by a more detailed description of them.

#	symbols	meaning
0		illegal
1	..RSTA,..SSTA	read or write job's starting address.
2	..RLFI	loaded file name (4 words: device, SNAME, FN1, FN2) (read only).
3	..RSTP	read DDT's symbol table pointer for this job. The left half is minus the length of the symbol table.
4	..RSYM,..SSYM	read or set value of a symbol.
5	..RJCL,..SJCL	read or clear the job's :JCL command.
6	..RPFILE,..SPFILE	read or set DDT's default :PRINT filenames.
7	..RSTB,..SSTB	read or write the whole symbol table.
10	..RCONV	read the symbolic equivalent of a number.
11 and 12		illegal (their old meanings were obsolete)
13	..RLJB	read the job number of the previously selected job.
14	..RRND,..SRND	read or set per-job miscelaneous flags.
15 & up		illegal

Types 1, 3, 11 and 12 read or write one word, at the address pointed to by the right half of the operation word. The other types read or write more words.

Type 1 reads or writes the start instruction, whose right half is the start address, and whose left half should be JRST or JUMPA. The start instruction may also be 0, meaning that there is no start address.

Type 2 stores 4 words starting where the right half points.

Type 3 stores a quantity whose left half is minus the size of the inferior's symbol table in DDT. The right half is the address in DDT of the symbol table, but it is a mistake to use that since DDT can shift the symbol table at any time.

Type 4 assumes, on read, that the right half points to a word with a SQUOZE symbol in it. If the symbol is defined, its value is stored in the location following the symbol. If the symbol is ".", the value of "." is returned. If the symbol is undefined, zero is stored where the symbol was and the following location is unaffected. A type 4 write defines the symbol pointed to by the right half to have the value specified in the location following it.

Type 5 allows an inferior to read its command string (set by :JCL <string> or :<prgm> <string>) from DDT. The job's .OPTION variable's bit 4.6 (OPTCMD bit) will be set by DDT if a command is available. If you don't try to get a command when that bit is off, you'll have no trouble being run by programs other than DDT. See "Running Programs", and the :JCL command, for information on how the contents of the command string are set. The string is transferred to the inferior as packed ASCII. The first word is always transferred. Successive words are transferred until either the previous word transferred was zero or the word about to be transferred into is nonzero. Note that the terminating character of the JCL will be either ^M, ^C or ^_, and that command string is not necessarily in upper-case. Type 5 write zeros the command string.

Type 6 reads or writes a block of 4 words as follows: device, SNAME, FN1, FN2 (all left-justified SIXBIT).

Type 7 is like ◊◊^Y, with the argument in the call used as the argument to the ◊◊^Y (ie as the address of the AOBJN pointer to the table to be replaced). Writing info type 7 is like doing a ^Y, with the arg to the call pointing to the AOBJN pointer to feed to the ^Y.

Type 10 provides the essential part of DDT's symbolic typeout. The argument is a number. It is replaced by the SQUOZE code for the symbol whose value is closest to but not larger than the number, or 0 if there is no such symbol. The word after the one containing the argument receives the difference between the argument and the value of the symbol, or the argument unchanged if there was no symbol.

Type 13 allows a program to find out what job was current when it was invoked. Thus, you can write programs which examine the data structure of the current job, as if they were DDT commands. Type 13 returns the job number of the job which was selected just before the one selected now. If you do this at random times (not just after being invoked with a colon) then that value has no significance.

Type 14 reads or writes a word of miscelaneous flags in DDT. At the moment, there is only one flag: bit 1.5, which, if set, means that :NOMSG 0 should be in effect while this job has the tty. Programs which print listings or graphs on terminals might want to set this bit to make sure that they are not spoiled by messages. Any unsolicited typeouts blocked by this flag are printed when next the job returns to DDT.

An illegal .BREAK 12, (this does not include undefined symbols) causes DDT to give a %PIILO interrupt to the inferior. For a block mode .BREAK 12, the AOBJN pointer is counted out and stored back.