FIELD

Friendly Integrated Environment for Learning and Development

Man Pages

Department of Computer Science
Brown University
Providence, RI, USA

Direct comments, questions, and bugs to:

ATTN: Software Librarian
Department of Computer Science
Box 1910, Brown University
Providence, R.I. 02912 USA

or electronically:

arpanet/csnet:	bwe@cs.brown.edu
uucp:	uunet!brunix!bwe

NAME

field, annotedit, annotview, formview, cbrowse, dbg, dbgview, flowview, viewevent, viewstack, viewtrace, xprof, xref, xrefview, – A friendly integrated programming environment for learning and development

SYNOPSIS

field [*objfile* [*corefile*]] [-options]

cbrowse [*objfile*] [-options]

flowview [*objfile*] [-options]

DESCRIPTION

FIELD is an open, integrated, UNIX-based programming environment that runs on top of X11. It offers a rich set of tools, both standard UNIX tools and tools developed at Brown, all with consistent visual interfaces in an integrated setting.

The environment can be run as a single system via the **field** command, as paired tools via the **dbgview** or **xrefview** commands, or as independent tools using any combination of commands. All tools that are run by a single user on a single workstation will communicate and interact with each other on the system being debugged. If **field** is run, then a control panel window is created from which the user can choose the desired tools.

The FIELD environment provides annotation editors for the source file for actual editing, for program viewing, and for interacting with the various tools. These editors consist of a text editor (EDT from the Brown Workstation Environment currently), and an annotation panel that allows annotations to be placed to the left of each line. Annotations are used to communicate from the source file to and from the other tools of the environment. Multiple annotation editors can be active at one time. Different types of annotation editors serve different purposes, by making different annotations available and by monitoring different sets of annotations. The **annotddt** tool provides a source interface for debugging. It offers annotations for setting break and trace points and will automatically change the file being displayed to show the current debugger focus. The **annotview** tool provides a viewer for other environment tools. It will change the file display to show various cross reference requests and compiler error messages. The **annotedit** tool provides an editor that will not change its focus automatically.

FIELD uses standard debuggers, either dbx or gdb, to control the execution of the system. A common textual interface to these debuggers is provided by the **ddt** front end which is used by FIELD. The command language here is similar to that used by dbx, with some gdb-like extensions and some minor modifications. While this interface can be used alone, in FIELD, it is generally used with the **dbg** tool. This tool provides a transcript of the ddt session as well as a set of user-definable buttons. The tool **dbgview** offers a split window that combines **dbg** with **annotddt** to provide a useful source-oriented debugger interface.

FIELD provides a full relational cross-reference facility. This facility, **xrefdb,** offers a textual relational algebra interface. This is used primarily by other tools within the environment. One such tool is the cross-reference interface **xref.** This provides for simple queries and displays the output in a text window. The user can click on the displayed output to cause an annotation editor to display the corresponding source location. This tool is also available as **xrefview** where it is paired in a split window with **annotview.**

There are also specialized, graphical viewers for the cross-reference database. **flowview** provides a view of the call graph. It allows the user to display all or portions of the program's call graph. The user can click on any node or arc to have an annotation editor display the corresponding source. The call graph can also be used to display program execution. The tool **cbrowse** provides a graphical browser on the C++ class structure of the user's application. This again supports hooks to an annotation editor and to displaying program execution.

Other tools are viewers that allow the user to see the current state of the program being debugged. The tool **viewstack** will provide a display of the execution stack that will be updated whenever the debugger gets control. The tool **viewevent** will provide a display of all the user's breakpoints. The tool **viewtrace** will show the current value of all variables or expressions being watched within the debugger. It can also be used to specify variables or expressions to watch.

Finally there are tools that provide visual interfaces to standard UNIX tools. **formview** offers a visual interface to a variety of versions of make including gnumake, sun make, and system 5 make. **xprof** provides a visual interface to the various profilers including prof, gprof, fgprof and iprof.

OPTIONS

The FIELD commands accept all the X11 options supported by the Brown Workstation Environment. These can be specified on the command line or in an X11 resource file. Resources can be specified for all field tools using the form field**resource*. Resources can be specified for an individual tool using the form field**tool***resource*. The resource file $HOME/.fieldrc is used by FIELD in addition to the user's current X resources. In addition to all resources provided by the individual components of the BWE toolkit (notably STEM and WIND), these include:

-background *color* or -bg *color*
: Set the background color.

-bordercolor *color* or -bd *color*
: Set the border color.

-borderwidth *width* or -bw *width*
: Set the border width.

-display *display*
: Set the display.

-font *font* or -fn *font*
: Set the default font.

-foreground *color* or -fg *color*
: Set the foreground color.

-geometry *geometry*
: Set the geometry of the first window.

-name *name*
: This option uses the given *name* as the command name. Running field with a command name of one of its tools is equivalent to just running that tool.

-style *style*
: This option lets the user specify the look and feel style that the FIELD tools will use. The style can be one of **3D** for the default 3-dimensional buttons, **2D** for 2-dimensional buttons, or **SIMPLE** for simplified 3-dimensional buttons.

-simple
: This option allows the user to run a simplified version of FIELD. This version assumes that there is only one source file for the executable. It provides fewer menus and buttons so as not to overwhelm the beginning programmer.

+query
: This option will cause FIELD to request that the window manager not query the user for the location of the windows.

In addition to these display options, all FIELD tools offer the following local options:

-auxd *auxd_file*
: The FIELD tools use the Brown Workstation Environment resource manager AUXD for defining many of their options, defaults, and internal workings. Normally, the default AUXD file for field is used. This file can be overridden using this option. Note that environment variables can be used to specify user-defined AUXD files that are specific to individual tools.

-msglock *lock_file*
: The FIELD tools communicate via a TCP/IP sockets through a central message server. The message server places its host name and port number in a lock file so that the various tools can find it.

Normally, that lock file is kept in /tmp (so its local to the given machine) and includes the user's name (so its specific to a particular user). This option allows the user to specify an arbitrary lock file. All FIELD tools that are run with the same lock file will communicate with each other; tools run with different lock files will not. This option, along with NFS, can be used to run different FIELD tools on different machines or to run multiple instances of FIELD on the same machine. The user should have write permission for the specified file.

X11 RESOURCES

In addition to the standard resources that can be specified on the command line, FIELD and BWE allow a number of private resources to be specified as a means of customizing the user interface. These can be defined in the user's .Xdefaults, in the file ˜/.fieldrc, or in the file specified by the FIELDRC environment variable. They include:

Resource	Format	Description
Aspect	x y	Aspect ratio
btn.background	color	Background color for buttons
btn.foreground	color	Foreground color for buttons
btn.gray	color	Gray color for buttons
btn.gray_mix	value	Background:foreground ratio for gray color
btn.gray_size	size	Size of gray region of 3D buttons in pixels
btn.on	color	On color for buttons
btn.on_mix	value	Background:foreground ratio for on color
btn.select	color	Select color for buttons
btn.select_mix	value	Background:foreground ratio for select color
btn.set_colors	flag	If true, then let user set button colors
btn.shadow_size	size	Size of shadow region of buttons in pixels
dialog.box_size	size	Size of check box in dialog in pixels
dialog.cursor_color	color	Color of cursor in dialog box
dialog.fontsize	size	Size of default dialog box font
dialog.textoff_color	color	Color of non-current text boxes
dialog.texton_color	color	Color of current text box
edt_mode	id	Mode for EDT editor bindings
gelo.data.bg	color	Background color for boxes in graphics displays
gelo.data.fg	color	Foreground color for boxes in graphics displays
leaf.titlefill	fill	ASH Fill pattern number for title region
MaxSize	x y	Maximum window size in pixels
MinSize	x y	Minimum window size in pixels
menu.border_x	size	X border size for sticky buttons
menu.border_y	size	Y border size for sticky buttons
menu.hi_color	color	Select color for sticky buttons
menu.lo_color	color	Disable color for sticky buttons
menu.norm_color	color	Color for sticky buttons
menu.scroll_width	size	Size of scroll region for sticky menu in pixels
move_mouse	flag	If set, mouse moved into dialog box automatically
pdm.background	color	Background color for pull down menus
pdm.border_x	size	X border size for pull down menus
pdm.border_y	size	Y border size for pull down menus
pdm.btn_font	font	Font for pull down menu buttons
pdm.foreground	color	Foreground color for pull down menus
pdm.ripoff	flag	Flag to allow/disallow ripoff menus
pdm.ttl_font	font	Font for pull down menu title bar
scroll.color	color	Scroll bar color
scroll.space	size	Size of sides of scroll handle
setup	style	WIND setup style (NO SETUP, BUTTONS, or SOLO)

show_active	flag	If set, keep icons for all active windows
show_unique	flag	If set, keep icons for unique windows
SizeInc	sz	Size increment
state.background	color	Default background for state menus
state.box_size	size	Size of check box for state menus
state.foreground	color	Default color for state menus

RESOURCE FILES

FIELD uses the AUXD resource manager that is part of the Brown Workstation Environment. It reads in an initial file that specifies the various resources and other files to be read in. The initial resource file is determined as follows:

1) If the -auxd option is specified, then that filename is used.

2) Let *name* be the name specified using the -name option or the tail of the command name used to invoke field if no -name option is specified. Let *NAME* be the upper-case translation of *name*. Then if the environment variable *NAME*.AUXD is set, then the filename specified there is used.

3) The file ˜/.*name*.auxd is used if it exists.

4) The file ˜/.field.auxd is used if it exists.

5) The file $PRO/lib/field/rundata/$ARCH/*name*.auxd is used if it exists.

6) The file $PRO/lib/field/rundata/$ARCH/field.auxd is used.

The heading FIELD in this AUXD file specifies the overall environment to be used by the system. It can contain the following items:

```
USE +
  NAME = pathname
  ENVIRON = environment_variable
```

These are additional files to use with AUXD at startup. The environment variable is used if it is defined. Otherwise the pathname is used. If the pathname does not begin with a ’/’ and does not exist, it is looked up in the FIELD rundata directory.

```
SETENV +
  NAME = name
  VALUE = value
  IFNDEF = .
```

These are environment variables sets to be done. The IFNDEF option will only cause this set to be done if the variable was not previously set.

```
LOAD +
  FILE = pathname
  CALL = routine
```

This allows arbirary routines to be dynamically loaded at startup. The entry point specified by CALL is called if the load is successful. This item can be used with only a CALL definition to cause routines in the bound version of field (or previously loaded items) to be called.

```
SERVICE +
  NAME = name
  CALL = routine
  SYSTEM = pathname
  ARGS = ( a1 ... )
```

This allows various services to be started at initialization. The service name is given by name. Then either the CALL field must specify a routine in field or that has been loaded, or the SYSTEM field must specify the system to be executed.

```
COMMAND +
  NAME = name
  CALL = routine
  DATA = string
  TWOARG = .
```

This item allows different commands to be run from the same binary. The NAME specifies the command name (i.e. annotview, dbgview, ...). The CALL field specifies the routine to be called in the binary. This routine is passed the root window as its first argument. The DATA field, if given, will be passed as the second argument. The files specified on the initial command line will be passed as the remaining arguments. Unless TWOARG is specified, only one such file can be named by the user.

```
BUTTON +
  NAME = name
  CALL = routine
  ICON_CHAR = char
  ICON_FONT = font
  MENU = menu
  ACTION = string
  KEY = keyname
  UNIQUE = .
  NOGENERIC = .
  NOTHREADS = .
  NOWINDOW = .
```

This item is used to define a command or window definition for the control panel by generating a WIND_DEF structure. The NAME field must be specified. Either the CALL field should name a routine or the ACTION field should contain a shell command to be executed. Either the ICON_CHAR and ICON_FONT fields should be set or the MENU field should be set if the button is to be displayed at any time. The UNIQUE flag insures only one such window exists at a time. The NOGENERIC flag means that the icon/window type is for internal use and no generic button for it will be made available. The NOTHREADS flag indicates that the window should not be started in a separate thread is Field is using threads. Finally the NOWINDOW field indicates that no window is to be created for this button.

```
STARTUP +
  SYSTEM = .      -- only execute if system given
  EDITOR = .      -- only execute if source given
  WAIT = .        -- pause until command terminates
  COMMAND = "command to exec"
```

This item allows the user to specify additional shell commands that should be executed when FIELD starts. The SYSTEM and EDITOR entries allow the command to be selective -- i.e. only execute when a system name or a source name are given on the command line respectively. The WAIT entry tells field to wait for the command to complete before continuing. The command string is executed within the users shell. In addition, the user can specify DEBUGGER and DBGEDITOR strings at the top level to specify alternates to dbg and annotddt when a system is given; and an EDITOR string to specify an alternative to annotedit when a source is given. The later actually setenv's ANNOT_EDIT (this is an alternative) unless the name begins with a '*'. If any of these is given as "*", then the corresponding tool is not run.

Other items in the AUXD file allow the user to customize the various FIELD tools. These are described in more detail in the manual pages for the individual tools. Note that the resource files are used for defining message patterns for the FIELD tools as well and care should be taken so that these definitions are preserved if private resource files are created for the various tools.

USAGE

The environment provided by FIELD consists of a number of tools that can either be invoked separately or that can be invoked as **field** using a control panel. The set of available tools will depend on the customization done by the user. This section describes the default field environment, assuming no user customization. The set of initially available tools that have icons on the control panel that can be invoked by clicking on them include:

annotview
: An annotation editor that will change file and line to view error messages and cross reference requests.

annotedit
: An general annotation editor.

annotddt
: An annotation editor that will change file and line to keep in sync with the debugger.

aedit An annotation editor without any annotations, i.e. a plain text editor.

dbg A visual interface to the DDT debugger. This uses the standard system debugger, generally dbx.

gdbg A visual interface to the DDT debugger using the GNU debugger gdb.

viewstack
: A stack viewer to be used in conjunction with the debugger.

viewevent
: A viewer showing currently active breakpoints and other debugger events.

viewtrace
: A viewer showing the current values of variables being traced.

formview
: A visual interface to make.

transcript
: A transcript window of makes done by the user. This is a simplified form of the formview command.

xprof A visual interface to prof, gprof, fgprof, or iprof.

xref A visual interface to the cross-reference facility (xrefdb) provided by FIELD.

fprof A visual interface to UNIX profiling.

typeedit An editor that allows the user to show how a data type should be displayed graphically.

display Graphical data structure display facility.

flow Call graph viewer.

cbrowse
: Class browser for C++ programs.

VT Separate window for program input and output. Normally program I/O is part of the debugger transcript. This tools allows it to be placed in a separate window.

In addition, on the *Commands* menu of the control panel, FIELD provides the following tools:

Help An interface to the hierarchical help facility of BWE. Note that the current set of help information for field is incomplete.

Mouse Help
: The mouse help window from the BWE help facility. It will attempt to dynamically show what the mouse buttons do.

Refresh Invokes the xrefresh command.

Directory
Lets the user set the working directory for all FIELD tools currently running.

Quit Exit from field.

The *UNIX* menu of the control panel contains miscellaneous system commands including:

Shell Invoke a UNIX shell using the EDT editor.

Memory Info
This command is for debugging purposes only.

There are several keyboard accelerators and functions defined within the FIELD environment. These include:

Key	Effect
Print	Used to dump window or screen to file or printer
Help	Requests help information
Meta-b	Build the default system
Meta-q	Quit out of FIELD

The *Print* key will cause a dialog box to come up asking the user to specify information on what window should be printed or saved, which of these should be done, and the formats to be used. After the dialog is accepted, the user should set things up for the print (i.e. wait for all refreshes to occur), and then hit the *Print* key again. The *Help* key, when pressed, will attempt to provide help information about the button or window that the cursor is over when the key is pressed. The current set of FIELD help information is somewhat incomplete however.

ENVIRONMENT VARIABLES

FIELD and the Brown Workstation Environment both use environment variables to allow the user to customize the environment. These include:

Variable	Default	Description
ANNOT_EDIT	annotedit	Default annotation editor to create
BUILD_RULES		User-specified default makefile
CC_COMMAND	CC	C++ 2.0 command name
DBX	/usr/ucb/dbx	Pathname for dbx
DDT_INIT	$HOME/.ddtinit	Initial command file for ddt
DISPLAY_DB	./.display_defs	Directory for saving data structure pictures
FIELDRC	$HOME/.fieldrc	Default FIELD resource file
GDB	/cs/bin/gnu/gdb	Pathname for gdb (Gnu debugger)
INCLUDE_PATH		Directories to search for include files in
MSG_GROUP		File to be used for message server host/port (see -msg option)
SOURCE_PATH		Directories to search for source files in
STDPASCAL		Do case mapping for standard pascal
USE_GDB		Use gdb rather than dbx if set
XRDB_FLAGS		-D and -U options for C++ compilation
FIELD_TMP	/tmp	Directory for shared temporary files
FORM_BACKEND	gnumake	Default make to use
ANNOT_AUXD		User AUXD file for annotation editing
BUILD_AUXD		User AUXD file for make interface
DBG_AUXD		User AUXD file for dbg debugger interface
CBROW_AUXD		User AUXD file for class browser
DISP_AUXD		User AUXD file for data structure displays
FLOW_AUXD		User AUXD file for call graph browser
BACKUPS	1	Number of backup versions of sources to save
EDT_BASIS_CMD		User defined editor command bindings

MOVE_MOUSE	Jump mouse into dialog box if set
SHOW_UNIQUE	Leave buttons for windows in control panel
NO_ICONS	Don't create icon buttons on control panel

FILES

FIELD is designed to be installed in subdirectories of a given host directory. At Brown, this is either /pro or /cs depending on the version of FIELD that is being used. In other installations, it may be an arbitrary directory. We will designate it $PRO. The architecture name (via the arch command on suns) is used where multiple systems must be supported from a common hierarchy. This is designated $ARCH.

$PRO/bin/field/*
$PRO/lib/field/help/*
$PRO/lib/field/rundata/$ARCH/*
$HOME/.ddtinit
$HOME/.fieldrc
$HOME/.edt.cmds
$HOME/.edtabbrev
$HOME/.field.auxd
$HOME/.*command*.auxd
$HOME/.dbgbtn
$HOME/.Buffers
./bBACKUP
./.dbgbtn
./.ddtinit
./.display_defs
./.field.auxd
./.fieldrc
.*.xref
.*.xrefrc
/tmp/msg.*hostname*.*userid*

SEE ALSO

annotedit(1), formview(1), cbrowse(1), ddt(1), display(1), flowview(1), viewevent(1), xprof(1), xref(1), xrefdb(1), userio(1), ddtfilter(1), msgserver(1), xrefserver(1), The Brown Workstation Environment Reference Manual.

COPYRIGHT

AUTHOR

Steven P. Reiss, Department of Computer Science, Brown University.

BUGS

Too numerous to mention. Please report any found to spr@cs.brown.edu so that they can be fixed.

NAME

annotedit, annotview, annotddt, aedit, codeview – FIELD annotation editors environment

SYNOPSIS

annotedit [*sourcefile*] [-options]

annotview [*sourcefile* [-options]

annotddt [*sourcefile* [-options]

aedit [*sourcefile* [-options]

codeview [*sourcefile* [-options]

DESCRIPTION

These commands invoke individual annotation editors as part of the FIELD environment. These can be used for editing, for program viewing, and for interacting with the other tools of the environment. They can be invoked either as separate tools via the above commands are from the FIELD control panel. These editors consist of a text editor (EDT from the Brown Workstation Environment currently), and an annotation panel that allows annotations to be placed to the left of each line. Annotations are used to communicate from the source file to and from the other tools of the environment. Multiple annotation editors can be active at one time. Different types of annotation editors serve different purposes, by making different annotations available and by monitoring different sets of annotations. The **annotddt** tool provides a source interface for debugging. It offers annotations for setting break and trace points and will automatically change the file being displayed to show the current debugger focus. The **annotview** tool provides a viewer for other environment tools. It will change the file display to show various cross reference requests and compiler error messages. The **annotedit** tool provides an editor that will not change its focus automatically. **codeview** provides a simple annotation readonly editor on a file with a single annotation that is displayed via highlighting for the current line being executed. It provides a more readable program visualization of the current line of execution than do the other annotation editors that use an iconic annotation for this purpose. Finally, **aedit** provides an editor that uses no annotations, i.e. a simple text editor with the same commands and interface as the other annotation editors.

OPTIONS

The options, X11 resources, files, and environment variables used by these commands are the same those used by the **field** command in general and the rest of the tools of the FIELD environment. See field(1).

RESOURCE FILES

The resource manager is used extensively by the annotation editor for defining both the different classes of annotation editors and the different types of annotations. The resource file entries for the annotation editor are grouped under the heading ANNOT. They allow the user to define different annotation editor types, to define new annotations, and to define buttons. The formats are as follows:

```
WINDOW +
  NAME = name
  PARSE = .
  USE = ( annotation types )
  SET = ( annotation types )
  SEARCH = ( annotation types )
  BUTTONS = ( button names )
  READONLY = .
  CREATE = .
  SENSITIVE = .
  CONTROL = .
  AUTOFILE = .
  MSG_OPEN = ( messages )
  MSG_CLOSE = message
  TITLE = title
  MODE = EDT mode
  CHOICEMENU = .
  NOMENU = .
  NOWINDOW = .
  NOMONITOR = .
  DISABLE = .
  LEVEL = level
```

This defines a new annotation editor type. The name is the internal name; the title is the string used as the window name. The USE set denotes the annotations that will be displayed by the editor; the SET list denotes the annotations that will cause the editor to change file or line; the SEARCH list denotes the annotations that explicit search buttons will be created for. The flag values include READONLY to make the editor readonly, CREATE to allow new files to be created, SENSITIVE to denote the type of sensitive region used in the annotation panel, CONTROL determines whether EDT saves a control file or not, AUTOFILE determines whether the editor should ask the user to set the initial file, and NOWINDOW denotes that the editor should run without displaying anything. MSG_OPEN lists messages sent when the editor is opened and MSG_CLOSE is the message sent when the editor is closed. The MODE parameter is passed to EDT to determine key bindings. The CHOICEMENU and NOMENU parameters allow the current annotation type selection to be made on a pull down menu or to be excluded from user selection respectively. Finally, the LEVEL parameter denotes the maximum level of annotations that will be displayed (although others in the USE list will be active).

```
ANNOTATION +
  NAME = name
  DISPLAY = text
  DISPLAY_FONT = font
  LEVEL = level
  MSG_ADD = message
  MSG_REMOVE = message
  MSG_SET = message
  MSG_UNSET = message
  MSG_UPDATE = message
  UNIQUE = .
  UNIQUE_WITH = annotation type
  IMMEDIATE = .
  SAVE = .
  PRIVATE = .
  ACCUMULATE = .
  QUERY = .
  MULTIPLE = .
  HILITE = .
  ALTERNATE = .
  JOIN = .
  TOP = .
  SPLIT_ONLY = .
  COLOR = color
  INFO = ( info name strings )
  KEY = "k" -- key to be used as shortcut
  PRIORITY = priority (default is 10)
```

This defines a new annotation. The annotation is displayed using the given text in the given font. Usually the font name is omitted and the text denotes an icon name (which can be defined using the ICON descriptor below). The annotation will be drawn in the specified color. The level denotes the annotation level. This is used along with the window level to determine if the annotation should actually be displayed. The MSG_ADD message is sent when the user tries to add the annotation; the MSG_REMOVE message is sent when the user tries to remove the annotation. The user can add or remove the annotation explicitly unless the PRIVATE flag is set. The MSG_SET message will cause the annotation to be added when it is received; similarly, the MSG_UNSET message will cause the annotation to be removed. The MSG_UPDATE string contains a message that will be sent for the given type of annotation whenever the file is saved. If UNIQUE or UNIQUE_WITH is set, then there can only be one instance of this annotation in the system. UNIQUE_WITH allows the specification of a group of annotations that are all unique with each other. A user request to add an annotation will not actually add the annotation unless IMMEDIATE is set. Setting SAVE causes the annotation to be saved with the file. Setting ACCUMULATE will cause multiple annotations at the same line of this type to have their text values concatenated and made into a single annotation. QUERY tells the editor that the user should be prompted for the field values specified in the INFO list when the annotation is added explicitly. Normally multiple annotations of the same type are ignored on a single line. MULTIPLE permits multiple anotations. HILITE, if set, will cause the annotation to select (and hence hilite) the corresponding editor line rather than putting up an icon in the annotation window. ALTERNATE will cause the editor to split the editor window and to use the bottom panel. JOIN will cause the editor to merge a split window. SPLIT_ONLY will cause the annotation to only be used if the window is currently split.

```
BUTTON +
 NAME = name
 OUTPUT = message
 WAIT_MSG = message
 SAVE = .
 WAIT = .
 FILE = .
 NO_FINISH = .
 FINISH1 = <string for <= 1 line of output>
 REMOVE = ( annotation types )
```

This defines a new annotation editor command. The name is the command name to be placed on the menu. The output is the message to be sent through the FIELD message server to actually execute the command. If either WAIT or WAIT_MSG is set, then the command will cause a dialog box to be put up while the command is executing. The WAIT_MSG determines a non-default string to be placed in this box. If SAVE is specified, then the file is saved before the command is executed. If REMOVE is specified than all annotations of the corresponding types are removed before the command is executed. FILE indicates that the return value of the command message is a filename that in turn contains the command output. This is used to display the result of the command.

```
QUERY +
 NAME = name
 OUTPUT = xrefdb output list
 EXPR = xrefdb query expression
 SEND = message to send
 DESCRIPTION = query descriptor string
 OUT_FORMAT = query output format string
 DISPFILE = . -- put display in separate file
 QUERYFILE = . -- put display in common query output file
 SETFILE = (file# line#) -- set file argument numbers
 DISPLAY = . -- force display of output
```

This defines an annotation command that is tied to the cross reference interface. It constructs a query using the OUTPUT and EXPR strings. Escape sequences in the EXPR string allow the inclusion of context-specific items. In particular, %L is replaced with the current line, %F with the current file, %E with the current editor selection, %S with the current system, and %d or %s with user-supplied parameters. Any of these can be of the form %'prompt'X to specify the prompt name to be given to the user. In addition, the string can be prefixed with %Q to indicate that the user shouldn't be queried for inputs unless necessary. The description string can have embedded %s's in it. These will be filled in using the parameters of the EXPR string (in the same order). The OUT_FORMAT string can also have embedded %s. These are replaced with the output values in the specified order. Specifying the output parameter numbers (1..n) for SETFILE will cause the editor to change files or lines after the query.

```
ICON +
  NAME = name
  XSIZE = size
  YSIZE = size
  DATA = ( integer value list )
```

This defines an icon, presumably to be used to denote an annotation. The list of integers should be ceil(XSIZE/32)*YSIZE long.

USAGE

The standard annotation editor window is divided into six panes (excluding the window decorations added by the WIND package of BWE). At the top of the window is a menu bar containing the titles of the pull dowm menus. The menus and buttons here are described below. Below this, to the left, is the annotation panel. This is where annotations are displayed for each line and where the user requests annotations. The center of the window is the actual display of the file. The scroll bar to the right of this display and the status window underneath are both part of the underlying EDT editor. Finally, the panel to the right lists the annotations that the user can create, highlighting the current default annotation.

This layout will vary based on the annotation editor chosen and the customization options that might be selected. If no annotations are available in the editor that would be displayed in the annotation window (as with **aedit** or **codeview**) then the annotation panel to the left is omitted. If there is only one annotation that the user can add, or if there are two and the customization specifies that the customization specifies NOMENU, or if the customization specifies CHOICEMENU then the menu of available annotations on the right will be omitted. In the later case, the available annotations will be placed on a pull down menu.

Mouse clicks in the annotation panel to the right can be used to add, remove or display annotations. Clicking with the left mouse button will either add or remove the primary annotation type, the one highlighted in the annotation panel to the right. The annotation will be removed if it already exists at the line and will be added otherwise. To explicitly add the annotation, even if an annotation of the same type exists, shift-click (or control-click or meta-click) the left mouse button. Note that adding an annotation does not always cause it to be displayed immediately. Breakpoint annotations will only be displayed when the debugger processes the request and the breakpoint is actually set. The right mouse button is handled similarly to the left but works with the alternate annotation type rather than the default. The alternate annotation type is not explicitly displayed, but is generally the second button on the annotation menu to the right or the previous value of the default annotation if pull down menus are used. The middle mouse button is used to request information on an existing annotation. Clicking in the window with this button will cause a dialog box to come up describing the leftmost displayed annotation on the given line. This leftmost annotation can be changed using the *Rotate* button on the *Annotate* menu when necessary. The dialog box displays whatever information the editor has about the annotation and gives the user the option of removing or resending the annotation if these operations are allowed. Note that shift-clicking in the editor window on a line can be used to add an annotation as well. In this case the left and middle mouse buttons will add the primary annotation and the right button will add the alternate annotation.

The primary and alternate annotation types (used above) are selected off the annotation menu on the right side of the annotation window. Clicking on a type with the left or middle mouse button will make it the primary annotation type. Clicking on a type with the right mouse button will make it the alternate annotation type.

The *Annotate* menu contains buttons for manipulating the annotations and the annotation editor in general. These include:

Set System
: Allow the user to set the system (object file and debugger) that this editor will talk to.

Remove All
: Allow the user to remove all annotations of one or more types.

Rotate
: Rotate the annotations of the line the editor cursor is currently at.

Resend
: Resend all annotations that support the RESEND option.

Display
: Set the display level of the editor. Annotations whose display levels are greater than the editor's display level will not be shown. Levels range from 0 to 1024.

Annotations
: Allow the user to select the set of annotations to be used or monitored by this annotation editor. Used annotations will be kept and displayed (if the level is correct); monitored annotations will cause the editor to change file and line to show a new annotation of the corresponding type when one is added.

Monitor
: This button allows the user to turn off monitoring of all annotations immediately. It can be toggled to restore monitoring to the currently monitored set of annotations.

Quit
: Exit the editor. If the file has been changed the user will be asked whether to save changes or not.

This and other menus supported by the annotation editor are dependent on the configuration. If the user specifies any annotation types under the SEARCH option in defining the editor type, then a *Search* menu will be present. This menu will contain two standard buttons and additional buttons for each annotation type specified with the SEARCH option. If no SEARCH option is specified, then the two standard search buttons will be placed on the *Annotate* menu. The buttons here are:

Search
: Search for an annotation of a given type after Requesting the type, a search direction, and whether to start the search at the current location or at the start of the file.

Search Again
: Repeat the previous *Search* starting at the current location.

Search for <type>
: Search for the next annotation of the specified type starting at the current location.

The *Select* menu will be used if the configuration requests CHOICEMENU. It will list the different annotation types that can be added by the user. Selecting a value here will make that value the new primary annotation and will make the previous primary annotation be the alternate annotation. The *Commands* menu will contain the buttons that have been defined for the editor in the resource file. The default buttons include:

Compile
: Save the file and send a request to the make interface to have it compiled.

Make
: Save the file and send a request to the make interface to have whatever systems it is a part of rebuilt.

Make Default
: Build the default system in the current directory.

SEE ALSO

field(1), The Brown Workstation Environment Reference Manual.

COPYRIGHT

BUGS

Too numerous to mention. Please report any found to spr@cs.brown.edu so that they can be fixed.

NAME

cbrowse – FIELD C++ class browser

SYNOPSIS

cbrowse [*objfile*] [-options]

DESCRIPTION

cbrowse is a graphical C++ class browser. It provides an interactive graphical interface to the xrefdb(1) cross referencer's information about the C++ class inheritance graph, classes and members. It allows the user to see a local section of the inheritance graph, the full graph, or to get detailed information about classes and members of the graph. It interacts with the FIELD environment, correlating items in the display with locations in the corresponding source files and using the display to show program execution if desired.

The class browser consists of one or two windows. The main window contains the visual display of the class information. A second, optional window provides an information display that offers a textual summary of the information about the currently selected class.

The class browser constructs the full class inheritance graph for the user's application using the xrefdb(1) facility. It then allows the user to selectively display the desired portions of the inheritance graph. There is a single class that is the currently selected class. This class is highlighted on the display and can be automatically enlarged. The display can be restricted to only show the inheritance graph above and below this class. Individual classes can be removed from either the global or a selective display.

The visual display of a class can be as simple as just showing the class name in a box, or as complex as showing the class name, all the members, public, protected and private, data and function, the friends, and links for the types of each member. Each of these various criteria can be individually displayed or not.

In the class display, the classes are shown as boxes of some type and the relations between them are shown as arcs. Public subclass links are shown as thick solid lines, private subclass links are shown as thin solid lines, friend links between classes are shown as dashed lines, and type links from members of the current class are shown as dotted lines. The class boxes, when fully displayed, consist of the class name at the top, an iconic region on the left denoting types of members, and, for each member, three fields, a state information field, the member name, and a link field to the corresponding type. The class name will be in a shaded region if the class is abstract. The icons on the left are either triangles or a box with an X. Triangles pointing to the left indicate data fields; triangles pointing to the right indicate function fields; solid triangles indicate private members; shaded triangles represent protected members; empty triangles indicate public members; the X box is used for friends.

A variety of different highlighting colors are used in the display. Light blue is used to denote the current class. Green is used to denote the currently selected member. Yellow is used to denote members that are inherited instances of the current member. Orange is used to denote the definition instance of the current member if that member is inherited. Thistle is used to indicate members that are along the inheritance path between this definition and the current member. Pink is used to indicate a member which is redefined by the current member. Cyan is used to indicate those members which redefine the current member. Finally, red is used to indicate a member that is currently executing if the browser is run in conjunction with the FIELD environment.

OPTIONS

The options, X11 resources, files, and environment variables used by these commands are the same those used by the **field** command in general and the rest of the tools of the FIELD environment. See field(1).

RESOURCE FILES

The class browser resource file can be used to define the default settings that determine how to display the class inheritance graph. The definitions include:

```
CBROW:
  METHOD = 0x### -- GELO method
  CONNMETHOD = # -- GELO connection method
```

FIXED = 0|1 -- use fixed size nodes
STANDARD = 0|1 -- use standard size nodes
CENTERED = 0|1 -- centered nodes
WHITE_SPACE = 0 .. 100
DISPLAY_ALL = 0|1 -- show all nodes or selected subset
DISPLAY_SUPER = 0|1 -- show superclasses
DISPLAY_SUB = 0|1 -- show subclasses
DISPLAY_LEVELS = -1 or 0... - levels to show in selected subset
DISPLAY_FORCE -- force update of display on new current
DISPLAY_SYSTEM = 0|1 -- show system classes
DISPLAY_NOEXPAND = 0|1 -- don't expand classes if true
DCLASS_SIMPLE = 0|1 -- simple display of classes
DCLASS_DETAIL = 0|1 -- show member details (on left)
DCLASS_ZOOM = 1 .. n -- zoom factor for current class
DCLASS_FRIEND = 0|1 -- show friend links
DCLASS_HIER = 0|1 -- show inheritance links
DCLASS_FIXCUR = 0|1 -- attempt to show all of current class
DMEMB_PUBLIC = 0|1 -- only show public members
DMEMB_DATA = 0|1 -- show data members
DMEMB_METHOD = 0|1 -- show function members
DMEMB_INHER = 0|1 -- show inherited members
DMEMB_LINKS = 0|1 -- show type links for current class
DMEMB_STATE = 0|1 -- show state information for members
DMEMB_FULLNAMES = 0|1 -- show full member names for inherited menbers

CLASS_EXCLUDE = (pat) -- regex list describing classes to omit
CLASS_INCLUDE = (pat) -- regex list describing classes to show
MEMBER_EXCLUDE = (pat) -- regex list describing members to omit
MEMBER_INCLUDE = (pat) -- regex list describing members to show

SELECT_STYLE = 1-9 -- hiliting for primary selection
MEMBER_STYLE = 1-9 -- hiliting for member selection
SUBMEMBER_STYLE = 1-9 -- hiliting for member in subtypes
SUPMEMBER_STYLE = 1-9 -- hiliting for member in supertypes
DEFMEMBER_STYLE = 1-9 -- hiliting for member definition
EVAL_STYLE = 1-9 -- hiliting for evaluting member
REPLACE_STYLE = 1-9 -- hiliting for member replaced by current

MEMBER_MOUSE = 0|1 -- show members under mouse

The GELO method and GELO connection method are integer values that can be derived from the $PRO/include/bwe/gelo.h include file. They determine the layout algorithms that are used in displaying the graph.

USAGE

The main **cbrowse** window is divided into two parts, a pull-down menu bar at the top and the actual graphical display below this. The menu bar contains two menus, *Browse* and *Display*. The *Browse* menu contains commands that apply to the brower as a whole. It includes:

Info Window

This button will cause a new window to be created on the current display. This window will be used by the browser to display more detailed information about the currently selected class in textual form. For more details on this window, see below.

Restart This button will discard all information about classes to be ignored and will then redraw the current display.

Update This button will request that **xrefdb** reload the current system and will then redraw the display. Any information about the current class or member as well as information about classes that shouldn't be displayed will be lost.

Set System
This allows the user to change the program that is viewed. This is effective both for viewing another system and for interpreting and sending messages to the other tools of the FIELD environment.

Quit This removes the browser window(s) and exits the browser.

The *Selection* menu contains options that allow the user to set and modify the current selection. The buttons here include:

Set Class
This causes a dialog box to be put up that allows the user to enter the name of the class to be viewed textually. If a new class is selected, the display will be updated accordingly.

Clear Class
This clears the current class selection. The display is then updated accordingly.

Select Class
This button causes a series of one or more dialog boxes to be put up that allow the user to go through and select the classes to be displayed. For each class there is an ignore button. Checking this button will cause the corresponding class to be omitted from further displays. The user can also select a particular class, thereby making it the current class. If there are more classes than fit in one dialog box, then the user will be provided with the option of going to the next (or subsequent or both as appropriate) set of classes.

Class Patterns
This button allows the user to select classes to be either included or excluded (or both) from the current display using regular expression patterns (regex (3)).

Member Patterns
This button allows the user to select the members to be displayed for all classes of for a given set of classes using a regular expression pattern.

The *Display* menu contains options that allow the user to customize the display. The buttons here include:

Options This button puts up a dialog box allowing the user to set any of the various display options other than those affecting the layout algorithms. The *Show all classes* option will force the display to include all the classes. The *Force redisplay on selection* option will cause the display to be updated whenever a new class is selected. This is the default as selecting a new class typically calls for a different display. The *Show superclass* and *Show subclasses* and the *Levels* options allow the customization of the display of the local hierarchy surrounding the given class. The option *Expand size of selected class* will attempt to make the current class large enough so that the member information can be read. The *Show friends* and *Show hierarchy* options specify what links (and associated classes) should be displayed. The *Show selection* option allows the user to expand the size of the selection to make it stand out. The remaining options specify which members should be shown. *Show member details* will cause the icons on the left to be drawn. *Show member links* will cause links to be drawn showing the types of the members of the current class. *Show member state information* will put up a state indicator to the left of the member name. This will contain one or more of the letters CSPVFI for constant, static, pure, virtual, friend, and inline respectively. Without the option *Show full member names* inherited members are just indicated with an initial colon-colon. *Show no members* will draw all classes other than the current one without members. *Only show public members*, *Show data members*, *Show function members* and *Show inherited members* all control the set of members to be displayed. Several of these options are individually settable using other buttons on this menu.

Show All
This option, if selected, will cause the browser to always show the full class inheritance graph as opposed to the local hierarchy for a given object.

Show Friends
This option will cause friend classes to be shown and friend links to be drawn.

Public Only
This will restrict the members that are displayed in a class to only the public members.

Show Data
This will cause the data members of a class to be included in the display.

Show Functions
This will cause the function members of a class to be included in the display.

Show Inherited
This will cause the inherited members of a class to be included in the display.

Layout This will put up a dialog box allowing the user to change or play with the layout algorithms that are used in drawing the class graph.

The mouse can be used within the graphics display for selecting the current class and member as well as related operations, all by clicking (no dragging). The operations that can be done this way are:

Click On	Button	Operation
Class	Left/Right	Make class current
	Middle	Show class information
	Shift-Left	Ignore class
	Shift-Right	Expand/contract class
	Shift-Middle	Show class details
Member	Left	Make class and member current
	Middle	Show member information
	Right	Make member current
	Shift-Left/Right	Make member current
Arc	Left	Make subclass current
	Middle	Show arc information
	Right	Make superclass current

Shift here indicates that either the shift, control or meta key was pressed while the mouse click occurred. In addition, any click done without a shift/meta/control will cause a message to be sent through FIELD requesting a source display of the associated line. For classes, this is the start of the class definition; for members, if the left or middle key is pressed it is the declaration of the member in the class and if the right key is pressed is the actual definition of the member; for relationships, it is the location where the relationship is established.

In addition to mouse clicks, **cbrowse** provides a few keyboard accelerators. These include:

Key	Operation
r	Redraw display
R	Reset and redraw display
c,C	Clear current class
a,A	Toggle show all mode
u,U	Update

The information window provides more detailed information about the current class. It is run as a readonly EDT editor. When a class is made current, the information for that class will be displayed. If the information was not previously displayed then it will be added at the end of the transcript being edited. The editor will automatically scroll so that the start of the information is at the top of the window. The user can click on any line of the display. Clicking on a line indicating a definition will cause a message to be sent requesting an annotation editor display the corresponding source. Clicking on a class name will make that class current and send an appropriate message. Clicking on a type or return type will make the corresponding class current and send a message. Clicking on a member will make the member current. It will also send a message requesting a source display of the declaration with the left or middle buttons and the definition with the right button. The mouse clicks here can be done using shift or meta if it is desired (plain clicks are also treated as editing operations).

SEE ALSO

field(1), xrefdb(1), annotview(1), The Brown Workstation Environment Reference Manual.

COPYRIGHT

BUGS

Too numerous to mention. Please report any found to spr@cs.brown.edu so that they can be fixed.

NAME

dbg, gdbg, dbgview, gdbgview – FIELD debugger interface

SYNOPSIS

dbg [*objfile* [*corefile*]] [-options]

gdbg [*objfile* [*corefile*]] [-options]

dbgview [*objfile* [*corefile*]] [-options]

gdbgview [*objfile* [*corefile*]] [-options]

DESCRIPTION

These commands form the workstation interface to the debugger. **dbg** provides an interface to the **ddt**(1) debugger provided with the FIELD environment. It offers both an EDT editor-based transcript window that allows textual interaction with ddt, and a user-definable set of buttons for frequently used commands. **dbgview** is an combined tool, merging **dbg** with **annotddt** to provide a debugger interface with an integrated source file window in a single tool. It splits its initial window in half and runs **dbg** in the top and **annotddt** in the bottom. The commands **gdbg** and **gdbgview** are similar to **dbg** and **dbgview** except that they use the Gnu debugger (gdb) rather than the standard system debugger dbx.

OPTIONS

The options, X11 resources, files, and environment variables used by these commands are the same those used by the **field** command in general and the rest of the tools of the FIELD environment. See field(1).

RESOURCE FILES

The initial set of buttons provided by the debugger interface is determined by the resource file. These are defined under the heading DBG as follows:

```
BUTTON +
  NAME = name
  OUTPUT = output string
  COLOR = color
  CONTINUE = .
  RUN = .
```

The NAME of each button is the text that will be displayed inside the button. The COLOR parameter determines the button color. If the CONTINUE flag is set, then the button will be highlighted whenever execution continues. If the RUN flag is set, then this button will be highlighted when execution begins. If no button has a CONTINUE flag, then a button with the RUN flag will also be highlighted when execution continues. At most one button should have a RUN or CONTINUE flag set. The OUTPUT string is the text that will be generated and sent to the debugger when the user hits the button. A newline character is automatically appended to the end of this text. The text can contain embedded control sequences that will be filled in. These include:

ˆ?	user's interrupt key
%Sreplace with current editor selection	
%L	replace with current line
%'name's	argument with given name
%F	replace with current function
%'name'd	integer argument with given name
%Q	don't ask user for values unless necessary

In general, if there is anything to be filled in in the output string, the user will be asked to confirm that the parameters are correct and will be allowed to correct them otherwise. The *%Q* sequence should appear at the start of the string if no confirmation is desired. The current line and function are those that the debugger is currently focused on. The name parameters on *%s* and *%d* are the prompts that will be placed in the dialog box when asking the user for the appropriate value.

The resource file can also contain the default font to be used for drawing the buttons under the heading BUTTON_FONT. It can also contain a flag value POPUP_ERROR. If this flag is set, then whenever the program being debugged halts because of a fault, a popup window will appear in the debugger interface informing the user. A related field is POPUP_SIGNAL which contains a list of integer signal numbers which this box should appear for (the default is all). It can also contain BUTTONS_PER_LINE, specifying the number of buttons to place on a single line in the button window.

USAGE

The debugger interface window is divided vertically into 5 parts. The top portion contains the pull-down menus. The second portion contains the transcript of the ddt debugger session in an EDT window. The third portion contians the mouse buttons that can be used in place of typing the associated commands. Beneath this are two lines, the first showing the current line of execution and the second showing the current debugger focus.

The pull down menu bar contains one menu for **dbg** and other menus that are associated with the EDT editor. The **Buttons** menu contains commands for manipulating the button panel part of the interface and the interface in general. It also provides an interface that allows the user to save a personalized button definition. It includes the buttons:

Remove This will pop up a dialog box asking the user which buttons to remove from the button panel.

Add This will pop up a dialog box asking the user to define a new button for the button panel.

Modify This will pop up a dialog box allowing the user to make changes to an existing button. Any aspect of the button, its name, color, output or position, can be changed in this way.

Reload This will reset the buttons as if the user started the debugger up again, i.e. using the user's saved button definitions. The user's button definitions are read in from the file ./.dbgbtn if this file exists, and ˜/.dbgbtn otherwise. If no user button file is found, then the default buttons found in the resource file are used instead.

Default This will reset the buttons ignoring any user definitions, i.e. using the definitions found in the resource file.

Save Local
This will save a copy of the current button definitions in the file ./.dbgbtn.

Save Global
This will save a copy of the current button definitions in the file ˜/.dbgbtn.

Quit This will cause the interface to exit.

The current version of **dbg** offers seven default buttons. These are:

Step Single step the current program.

Next Single step the current program, skipping over subroutine calls.

Continue
Continue execution.

Run Rerun the current system using the previous set of arguments.

Stop Interrupt execution of a running system.

Kill Terminate the current running system.

Print It Print the current editor selection. To use this the user should first pick an expression in any EDT editor window on the screen.

For information on the commands and the textual interface offered by **dbg**, see ddt(1).

SEE ALSO

annotddt(1), field(1), ddt(1), dbx(1), The Brown Workstation Environment Reference Manual.

COPYRIGHT

BUGS

Too numerous to mention. Please report any found to spr@cs.brown.edu so that they can be fixed.

NAME

ddt, gddt – FIELD debugger interface

SYNOPSIS

ddt [*objfile* [*corefile*]] [*-g*] [*-C*] [*-s*] [*-l*] [*-f*] [*-c* context] [*-n*]

gddt [*objfile* [*corefile*]] [*-C*] [*-s*] [*-l*] [*-f*] [*-c* context] [*-n*]

DESCRIPTION

ddt is an interface for FIELD to the system debugger. It provides a consistent interface to either dbx or gdb on a variety of different machines. It provides and handles messages for the FIELD environment. It does name mapping (mangling and demangling) for both C++ (AT&T 2.0) and Berkeley Pascal.

ddt offers a rich command set. This is based primarily on the syntax of dbx, with several useful gdb extensions. The primary difference from a standard dbx syntax is that dbx uses the trace command for tracing both lines, functions and variables while **ddt** uses trace to trace lines and functions and the watch command to trace variables.

ddt operates by actually running the system debugger (either dbx or gdb) in a separate process and communicating to it via a pty. It can either be run in a standalone mode or as part of the FIELD environment. It is run by the FIELD tools dbg, dbgtool, gdbg and gdbgtool. **gddt** is a version of **ddt** that will run the gnu debugger gdb by default.

OPTIONS

-g
This option will cause **ddt** to run gdb. **ddt** -g is equivalent to **gddt**.

-C
This option forces **ddt** to run in C++ mode, meaning that C++ name mangling and demangling will be performed. Normally **ddt** will determine the mode by reading the symbol table of the program being debugged.

-s
This option places **ddt** into a mode suitable for standard Pascal (i.e. Berkeley Pascal with the -s or -l options) where most names are mapped to lowercase.

-l
This option places **ddt** in local mode. This is the default. In local mode the system does not use the FIELD message facilities and cannot be used with other tools of the FIELD environment. However, multiple copies of **ddt** can be run without interfering with each other and performance is improved.

-f
This option places **ddt** into FIELD mode. This is needed when ddt is to be run as part of the FIELD environment. It is used implicitly when ddt is run using any FIELD tool.

-c *context*
This option allows the specification of a separate debugging context. This is a subcomponent of the system name and, when fully implemented, will allow the debugging of separate threads of control from separate front ends.

-n
This option causes **ddt** to run as a front end only. Instead of interpreting commands, the interface in this case will map the commands to messages for the FIELD message server. This assumes that some other copy of **ddt** is already running to handle these messages.

USAGE

The command language of **ddt** is borrowed mainly from the UNIX debugger dbx with some commands taken from from gdb. The whole command set includes:

Execution and Tracing

args	break	catch	cont
delete	display	handle	ignore
monitor	next	nextnext	rerun
run	return	status	step
stop	stopexit	trace	update
watch	when		

```
Displaying and Naming Data
   assign    call      down      dump
   info      print     printf    up
   whatis    where     whereis   which
   #

Accessing Source Files
   cd        file      func      list
   unuse     use       /         ?

Programming
   define    exec      if        loopexit
   msgf      undefine  vset      vsetq
   while

Miscellaneous
   alias     context   debug     help
   kill      make
   printenv  pwd       quit      quote
   setenv    sh        system    xset

Machine Level
   nexti     stepi     stopi     tracei
   watchi
```

The individual command formats are:

run

```
     run                 - Begin execution of the program with the current
arguments
     run <args>          - Begin execution of the program with new arguments
```

rerun

```
     rerun               - Begin execution of the program with no arguments
     rerun <args>        - Begin execution of the program with new arguments
```

args

```
     args                - Use no arguments for next run command
     args <args>         - Use given arguments for next run command
```

trace

```
     trace [ <if cond> ]                 - Trace each source line
     trace in <proc> [ <if cond> ]       - Trace each source line while in proc
     trace <line#> [ <if cond> ]         - Trace execution of the line
     trace at <line#> [ <if cond> ]      - Trace execution of the line
     trace <proc> [ <if cond> ]          - Trace calls to the procedure
     trace in all                        - Trace all procedure calls
     trace in all <pat> [ <if cond> ]    - Trace all functions matching <pat>
     trace all                           - Silently trace each source line
                                         - NOTE: Optional <if cond> causes
tracing
                                           to be performed only when
                                           condition <cond> is true as
trace
                                           point is reached
                                         eg:   trace 100 if i == 5
```

watch

```
   watch <exp> at <line> [ <if cond> ]  - Print <exp> when <line> is reached
   watch <var> [ <if cond> ]            - Trace changes to the variable
   watch <var> in <proc> [ <if cond> ]  - Trace changes to variable in
procedure
                                   - NOTE: Optional <if cond> causes
tracing
                                        to be performed only when
                                        condition <cond> is true as
trace
                                        point is reached
                                    eg:   trace 100 if i == 5
```

update

```
   update at <line> [ <if cond> ]  - Update views when <line> is reached
   update in <proc> [ <if cond> ]  - Update views when <proc> is entered
   update                         - Update views whenever debugger can
   update in all                   - Update on entry to all fcts
   update in all <pat>             - Update views for all fcts matching <pat>
   updateq at <line> [ <if cond> ] - Quietly update views when <line> is
reached
   updateq in <proc> [ <if cond> ] - Quietly update views when <proc> is
entered
   updateq                        - Quietly update views whenever debugger
can
   updateq in all                  - Quietly update on entry to all fcts
   updateq in all <pat>            - Update views for fcts matching <pat>
```

stop

```
   stop at <line> [ <if cond> ] - Stop execution at the line
   stop in <proc> [ <if cond> ] - Stop execution when <proc> is called
   stop in all                 - Stop whenever a routine is called
   stop in all <pat>           - Stop in all functions matching <pat>
   stop <var> [ <if cond> ]     - Stop when value of <var> changes
                              NOTE: Optional <if cond> causes execution to
                                    stop only if condition <cond> is true
                                    when appropriate stopping point is
reached
                              eg:   stop at 100 if i == 5
   stop <if cond>              - Stop if condition true
   stop                       - Stop execution (interrupt)
```

break

```
   break at <line> [ <if cond> ] - Stop execution at the line (temporary)
   break in <proc> [ <if cond> ] - Stop execution when <proc> is called
   break in all                 - Stop whenever a routine is called
   break in all <pat>           - Stop in all functions matching <pat>
   break <var> [ <if cond> ]     - Stop when value of <var> changes
                              NOTE: Optional <if cond> causes execution
to
                                    stop only if condition <cond> is true
                                    when appropriate stopping point is
reached
                              eg:   stop at 100 if i == 5
```

break <if cond>   - Stop if condition true

stopexit

stopexit in <proc> [<if cond>] - Stop execution at return of proc
stopexit in all   - Stop execution at all returns
stopexit in all <pat>   - Stop at return of all functions matching <pat>

step

step   - Single step one line (step INTO calls)
step <n>   - Single step <n> lines (step INTO calls)
step in <proc>   - Continous single step in procedure
stepq in <proc>   - Quietly continuous single step in procedure

next

next   - Step one line (skip OVER calls)
next <n>   - Step <n> lines (skip OVER calls)
next in <proc>   - Continous single step in procedure (step OVER calls)
nextq in <proc>   - Quietly continuous single step in procedure

status

status   - Print trace's, when's, and stop's in effect

delete

delete <number> ...   - Remove trace's, when's, or stop's of given number(s)
delete all   - Remove all trace's, when's, and stop's
delete in <proc>   - Remove all events in given procedure
delete at <line>   - Remove all events at given line

handle

handle   - Print status of signals
handle <sig> ignore   - Ignore given signal
handle <sig> catch   - Catch given signal
handle <sig> printuse - Tell when signal occurs and use it
handle <sig> printign - Tell when signal occurs and ignore it

ignore

ignore   - Print status of signals
ignore <sig>   - Ignore given signal

catch

catch   - Print status of signals
catch <sig>   - Catch given signal

cont

cont   - Continue execution
cont <sig>   - Continue execution with signal

return

return   - Continue until return (Not available on Suns)
return <proc>   - Continue until return from proc (ibid for suns)

call

call <proc>([params]) - Call the procedure

assign

assign <var> = <exp>   - Assign the value of the <exp> to <var>

print

```
print <exp> ...        - Print the value of the expression(s) <exp> ...
```

printf

```
printf "format", <exp> ...  - Print the value of the expression(s) <exp>
msgf "format", <exp> ...  - Print the value of the expression(s) <exp>
```

up

```
up                    - Move up the call stack one level
up <number>           - Move up the call stack <number> levels
```

down

```
down                  - Move down the call stack one level
down <number>         - Move down the call stack <number> levels
```

file

```
file                  - Print the current context
file <filename>       - Change the current file
```

func

```
func                  - Print the current context
func <proc>           - Change the current function to function
                      or procedure <proc>
```

where

```
where                 - Print a procedure traceback
where <num>           - Print the <num> top procedure in the traceback
where <top> <bot>     - Print the <top> top procs and <bot> bottom
```

dump

```
dump                  - Print a procedure traceback with locals
dump <top>            - Dump <top> top procedures
dump <top> <bot>      - Dump <top> top and <bot> bottom procedures
```

list

```
list                  - List 10 lines
list <first>, <last>  - List source lines from <first> to <last>
list <proc>           - List the source to <proc>
```

use

```
use                   - Print the directory search path
use <dir> ...         - Add to the directory search path
```

unuse

```
unuse                 - Clear the directory search path
```

kill

```
kill                  - Kill the process being run
```

quit

```
quit                  - Exit from the debugger
```

debug

```
debug                 - Print the name and args of the program being
debugged
debug  prog [core]    - Begin debugging <prog>
debug * [core]        - Begin debugging same system, new core
```

cd

```
cd <dir>              - Change the current <dir>
```

tracei

```
    tracei <address> <if cond>      - Trace execution of location <address>
```

stopi

```
    stopi at <addr> [ <if cond> ] - Stop execution at location <addr>
    stopi <var> [ <if cond>      - Stop execution when value of <var> changes
                                 NOTE: Optional <if cond> causes execution
  to
                                    stop only if condition <cond> is true
                                    as appropriate stopping point is
  reached
                                 eg:   stopi at 0x1017 if i == 5
    stopi <if cond>              - Stop if condition true
```

watchi

```
    watchi <var> <at addr> <if cond> - Trace changes to <var> at <address>
                                 - NOTE: Optional <if cond> causes tracing
  to
                                      be performed only when condition
                                      <cond> is true as trace point is
                                      reached
                                   eg:   tracei 0x1017 if i == 5
```

monitor

```
    monitor <exp> at <line> <if cond> - Trace value of expression
    monitor <exp> [ in <proc> ]      - Trace value of expression
```

whatis

```
    whatis <exp>         - Print the type of <exp>
```

whereis

```
    whereis <name>       - Print all declarations of <name>
```

which

```
    which <name>         - Print full qualification of <name>
```

quote

```
    quote <command>      - Pass command to dbx
```

info

```
    info so <pattern>     - Print all sources matching pattern
    info fu <pattern>     - Print all functions matching pattern
    info va <pattern>     - Print all variables matching pattern
    info ty <pattern>     - Print all types matching pattern
    info fi               - Print current system/core file names
    info ru               - Print current run arguments
    info dy <expression>  - Print dynamic type of C++ expression
```

while

```
    while <expr>          - Repeatedly execute commands
      <commands>               (see loopexit)
    end
```

if

```
    if <expr>             - Conditionally execute commands
      <commands>
    [ else
      <commands>
    end
```

loopexit
```
loopexit             - Exit from innermost while
```
system
```
system id            - Set system name
system id int        - Set system and context
```
make
```
make                 - Request make of system
make <name>          - Request make of given item
```
printenv
```
printenv             - Print the current environment
printenv <name>      - Print the name in the environment
```
pwd
```
pwd                  - print current working directory
```
sh
```
sh <command>         - Execute the command in the shell
```
help
```
help                 - Provide general help
help <cmd>           - Provide help on given command
```
display
```
display <exp>        - Display expression at each stop point
```
stepi
```
stepi                - Single step one instruction
stepi <count>        - Single step <count> instructions
```
nexti
```
nexti                - Single step one instruction over calls
nexti <count>        - Nexti <count> times
```
xset
```
xset <variable> <value>   - Set internal DDT variable
                          - The valid variables and values are:
     FORCE_RUN      0 or 1    Run even if out of date (default)
     CPLUSPLUS      0 or 1    Do CC name mapping
     STDPASCAL      0 or 1    Do pc -s name mapping
     STOP_UPDATE    0 or 1    Send update message at each stop
     STACK_TOP      0 or n    Display top n frames (all)
     STACK_BOTTOM   0 or n    Display bottom n frames (all)
     STACK_GLOBAL   0 or 1    Display globals with stack
     STACK_NO_MAIN  0 or 1    Skip main on stack (for Pascal)
     IGNORE_TEMPS   0 or 1    Ignore C++ temporary names on stack
```
/
```
/pattern/            - Search forward for pattern
```
?
```
?pattern?            - Search backward for pattern
```
#
```
<address> / <format>          - dump address in given format
<address> / <count> <format>  - dump count starting at address
```
when
```
when at <line> [ <if cond> ] <cmds> - execute cmds at given line
when in <proc> [ <if cond> ] <cmds> - exec cmds at procedure entry
                                    -  cmds can be { ... } or on
                                    -  separate lines
```

define

```
define name             - define procedure
  commands
end
```

undefine

```
undefine name           - undefine procedure
```

vset

```
vset name = expression     - set name to evaluated expression
```

vsetq

```
vsetq name = expression    - set name to unevaluated expression
```

exec

```
exec name arg1,...,argn    - execute procedure with args
```

ENVIRONMENT VARIABLES

GDB This specifies the pathname to be used in running the gdb debugger. The default is probably only suitable for use at Brown. It is /cs/bin/gnu/gdb.

DBX This specifies the pathname for running the dbx debugger. The default is /usr/ucb/dbx.

DDT_INIT
This specifies a file that will be read in as part of **ddt**'s initialization. It typically includes commands to customize an appropriate debugging environment.

USE_GDB
If this variable is set then **ddt** will use gdb rather than dbx as the default debugger.

STDPASCAL
Setting this variable is equivalent to running the system with the -s option.

CPLUS20
Setting this variable is equivalent to running the system with the -C option.

FILES

FIELD and **ddt** are designed to be installed in subdirectories of a given host directory. At Brown, this is either /pro or /cs depending on the version of FIELD that is being used. In other installations, it may be an arbitrary directory. We will designate it $PRO. The architecture name (via the arch command on suns) is used where multiple systems must be supported from a common hierarchy. This is designated $ARCH.

$PRO/lib/field/rundata/$ARCH/ddt.help
$HOME/.ddtinit
./.ddtinit

SEE ALSO

field(1), dbg(1), dbx(1), gdb(1).

COPYRIGHT

BUGS

Too numerous to mention. Please report any found to spr@cs.brown.edu so that they can be fixed.

NAME

ddtfilter – C++ name demangler

SYNOPSIS

ddtfilter [-cX] [infile [outfile]]

DESCRIPTION

ddtfilter is a C++ (AT&T 2.0) name demangling program that uses the name demangler built for the FIELD ddt(1) debugger. It scans the input file (either stdin or *infile*), converting all C++ mangled names to their full demangled equivalent and writes the result to the output file (either stdout or *outfile*). The input file can be specified as - to denote stdin.

OPTIONS

-cX This option, where X is an arbitrary character, will cause any spaces in the demangled names to be replaced by the character X.

SEE ALSO

ddt(1)

COPYRIGHT

BUGS

Too numerous to mention. Please report any found to spr@cs.brown.edu so that they can be fixed.

NAME

display, disptype, typeedit – FIELD data structure viewer

SYNOPSIS

display [*variable*] [-options]

disptype [*variable*] [-options]

typeedit [*typename*] [-options]

DESCRIPTION

These three commands form the FIELD data structure display facility. They provide an interface to the GELO component of the Brown Workstation Environment, mapping C, C++, and Pascal data structures from the form used by ddt to the proper displays. They allow the user to define the display format interactively as well as provide different default display formats for a variety of structures. For these tools to operate correctly, there must be an active FIELD debugger.

The command **display** provides a display of a single user data structure. The structure name can either be given on the command line or will be requested interactively when the display starts.

The command **typeedit** provides an editor that allows the user to specify the display format(s) for a given type. The type can either be given on the command line or will be requested interactively.

The command **disptype** is a combined FIELD tool. It splits its initial window in half, running the display facility in the top half on the specified or requested data structure and runs **typeedit** in the bottom half on the type of this data structures.

OPTIONS

The options, X11 resources, files, and environment variables used by these commands are the same those used by the **field** command in general and the rest of the tools of the FIELD environment. See field(1).

RESOURCE FILES

The **display** and **typeedit** resouorces are kept in resource files under the heading DISP. The following components can be defined:

```
BUILD_CLEAR = .
SIMPLE = .
```

The BUILD_CLEAR flag indicates that whenever a system is reloaded by the debugger, (typically after the system has been rebuilt,) that saved type information should be cleared. This is a little slower than not clearing, but insures that any changes made to datatype definitions are reflected in the display. This is set in the standard resource file. The SIMPLE flag is another method (in addition to -simple on the command line and *simple in the X11 resource file) of specifying that simplified menus should be used.

USAGE

The menus and interfaces provided by the data structure display facility, especially the type editor, are probably the most complex and confusing in the FIELD environment.

The **display** package, if run from FIELD or if started without a variable name puts up a dialog box that lets the user specify the variable to display. In addition to specifying the variable name, the box allows the user to specify the file and routine of that variable to distinguish between variables of the same name. It also allows the specification of the system being debugged that this display is talking to. Finally, the dialog box allows the user to specify the default display types. Two options are available. The *Nested display* option lets the user switch between a default display consisting of nested boxes and a display consisting of boxes and arrows. The *Use database* option specifies that previously saved type definitions for this directory should be used. This information is kept in a subdirectory *.display_defs* in the current directory.

Once the **display** tool finds an acceptable variable name, it segments its window. The top of the window contains a title bar with the variable name and type. The user can click in this window with the mouse, requesting that the above dialog box be put up so that the variable being displayed can be changed. Below this title bar is a menu bar containing the pull down menus for the display package. These are derived directly from the PEAR component of BWE. The main display is below this. To the right and below the main display are scroll bars. These are used to pan over the display after zooming has occurred.

The user can use the mouse to select items in the display. The left mouse button is used to make the primary selection; the middle mouse button is used to make the secondary selection. The right mouse button makes a tertiary selection. Currently there is no use for a tertiary selection. In each case, multiple clicks at the same location will select up the display hierarchy and a second click (at a different location) in the same object will deselect it. On a color display, the primary selection is blue, the secondary green and the tertiary yellow. On a monochrome display the primary selection is inverted, the secondary selection is dark gray, and the tertiary selection is light gray.

The *Edit* pull down menu contains buttons that allow editing of values in the display. The buttons here include:

Set Value
: This sets the value of the primary selection. If there is no primary selection then this button is ignored. If there is a secondary selection, then the value of the primary selection is set to that of the secondary selection. Otherwise, a dialog box is put up requesting that the user specify a value and giving several options based on the expected datatype. Any value here should be in a format acceptable to the debugger.

Copy Value
: This is identical to *Set Value*.

Set Contents
: This is identical to *Set Value*.

Copy Contents
: This is identical to *Set Value*.

Delete This sets the value of the primary selection to 0 or an appropriate NULL value.

New Value
: This is identical to *Delete*.

Expand This requests a new value from the user for the primary selection.

Make Current
: This is a no-op.

Create This is a no-op.

Quit This closes the window and exits the tool.

The *Layout* menu contains additional editing commands for manipulating graphs. None of the buttons on this menu are currently functional.

The *Display* menu contains commands for controlling the display. It contains the following buttons:

Update This button causes the display tool to go back to the debugger and reconstruct the value of the structure being displayed. It also reverts to a non-zoomed, no-selections, display of that structure.

Emphasize
: This command takes the primary selection and makes it larger if possible, shrinking the rest of the display as necessary to accomodate the new size.

Deemphasize
: This command takes the primary selection and makes it smaller if possible, enlarging the rest of the display as necessary to accomodate the new size.

Zoom In
: This commands zooms the display to show the central portion of the current window. After zooming, the scroll bars on the right and bottom can be used for panning. Every second zoom will

cause the display tool to recompute items that were too small to display initially.

Zoom Out
This command zooms out.

Heuristics
This command allows the user to choose among a large set of heuristics for graph layout. The primary selection should be the layout that the heuristics are to apply to.
For more information see the GELO manual.

Show Item
This button takes the primary selection and makes it be the top item being displayed.

Show All
This button causes the value of the specified variable to be the current item being displayed. It undoes the effect, for example, of a *Show Item*.

Match This button asks the user to set a match string for the top level object. Match strings are used to specify different drawing methods for types. Thus, this button allows the user to choose between different ways of drawing a type provided these ways have been set up correctly with the type editor.

Clear Choices
This button clears all current selections.

The *Inset* menu is used to control an inset panel that can be used in conjunction with the data structure display. This window, originally in the lower right quadrant of the main display, can be used to show more detail of some item being displayed. The buttons on this menu include:

Show Item
This takes the primary selection and places it in the inset window. The inset window will be made visible if necessary. It will redraw the item as a top level item, thus the display may differ from the nested display of the original selection.

Hide Inset
This makes the inset window invisible.

Show Inset
This makes the inset window visible. If nothing was previously displayed in the inset window, then this button will cause the primary selection to be displayed there.

Clear This clears the display in the inset window, causing it to display a NULL value or a NULL type.

Refresh This will refresh the inset window.

Resize This will ask the user to rubberband a box to show where the inset window should be placed within the parent display.

New Inset
This button is a no-op.

The *Eval* menu contains miscellaneous buttons, most of which do not apply to the data structure display tool. The only buttons that are active here are:

Reload Types
This button will cause all type definitions to be recomputed, either from saved user files or by requerying the debugger for type information. If the BUILD_CLEAR flag is not set in the resource file, then this button should be used whenever the datatypes being displayed change and the system is reinitialized in the debugger.

The display package also allows a number of keyboard accelerators. These are either equivalent to mouse clicks at the current position or to selecting the appropriate menu buttons. They include:

Key	Menu

1	
2	
3	
d,D	Delete
e	Emphasize
E	Deemphasize
h,H	Hide Inset
n,N	New Value
s,S	Show Item
t	Set Contents
T	Copy Contents
v	Set Value
V	Copy Value
u,U	Update
z	Zoom In
Z	Zoom Out

The **typeedit** tool is used for describing and editing the display formats used by the **display** tool. When it is started without a type name, or when the user requests a new type to be edited, it puts up a dialog box asking the user to specify the type. The user must provide a type name. This should be the same name that **display** shows when displaying the variable in question. The user can optionally provide a file and function name to further qualify which definition of that type is desired. Two additional options can be specified as with **display**:, the type of default display and whether the local database of type definitions should be used. If the database is used, it is used both for retrieval and for storage.

Once a type is correctly specified, the **typeedit** window is subdivided into a set of panels. The title bar at the top contains the type name currently being edited. The user can click on this title to request that a different type be edited. Below this is a menu bar for the available pull down menus. Below this to the right is a menu box containing options that can be used for display. This includes a number of standard options and all the fields of the type being edited. To the left of this is a pictoral representation of the resultant type format. Finally, below this representation box is a small box labeled *Include* that is also used as part of the representation.

Types are displayed using the GELO package of the Brown Workstation Environment. This package provides for hierarchical displays. Each level of the hierarchy can be an object of a given flavor. It can be a box object, in which case it is displayed as a rectangular region possibly containing a shape and text. It can be a tiled object, in which case it is displayed as a rectangular region subdivided into non-overlapping rectangular tiles. Each of these tiles can contain a value, either displayed according to the type of the value, displayed as a box, displayed as a graph, or displayed as an arc to another item. The displayed object can be an arc within a graph. It can also be a layout, that is a arbitrary graph with values being used to designate its contents, typically nodes and arcs.

Each type is specified as a series of top level objects along with conditions. The display for a type is determined by going through this series in order and selecting the top level object that satisfies all conditions. This top level object determines how the resultant user data of this type is to be drawn. Any components of this top level object are then filled in recursively. The conditions that can be used to differentiate types include whether the user data is NULL, a match string, whether the display is occurring within a layout, whether the display is at the top level or is nested, and whether this display is occurring within an array. (An additional option, based on a user-specified condition, is currently not implemented.) Match strings are arbitrary strings that can be specified either as part of the editing process or by the user to distinguish different drawing strategies. They are typically inherited through the hierarchy.

The pull down menu contains menus that allow the user to do much of the editing. Additional editing is done with the mouse in the representation window, in the menu to the right, or in the Include box at the bottom. Before additional editing can be done, the nature of the top-level object for this condition and the type being edited must be selected. The effect of the additional editing is determined by the flavor of this top level object.

The *File* pull down menu contains general options for the type editor. It includes:

Finer Grid
: This doubles the number of grid lines that are used in constructing rectangular tilings for tile-flavored objects.

Courser Grid
: This halves the number of grid lines that are used in constructing rectangular tilings for tile-flavored objects.

Show Grid
: This turns the grid display on or off for tile-flavored editing.

Clear This clears all information about the current type. It removes all alternatives.

Remove This removes the current alternative for the type being edited.

Save This saves the current type information. This command must be issued before any editing changes will effect a data structure display. If the database is used, then this command will also write the type display definition out to the database directory.

Restart This will reload the type definitions from the database or will effect a Clear if no database is used.

Quit This will terminate the type editor.

The *Top* menu is used for selecting the flavor of the top-level object and of selecting conditions for this type display. It includes:

Default This will load the default ways of drawing the type into the list of alternatives. Several default alternatives are automatically created for a type if no user-provided definition exists. This command allows the editor to explicitly specify these alternatives, thereby allowing the user to modify them rather than starting from scratch.

Tiled This makes the top level object tile-flavored.

Layout This makes the top level object layout-flavored.

Box This makes the top-level object box-flavored.

Arc This makes the top-level object arc-flavored. It also implicitly adds the condition for this alternative that the display must occur in a layout. Note that adding an arc between two items will automatically force those two items to be displayed.

Ignore This is an alternative to displaying an object. If this alternative is chosen, then nothing is displayed for the object.

Next Alternative
: This displays the next (subsequent) alternative in the series of alternatives.

First Alternative
: This displays the first alternative in the series of alternatives.

New Alternative
: This creates a new alternative BEFORE the current alternative. It puts up a dialog box asking the user to specify the conditions for the new alternative.

Set Conditions
: This puts up a dialog box allowing the user to set or edit conditions for the current alternative.

The *Edit* menu is used to select the flavor for components of a tiled object. It can only be used after a tile component has been selected. (See discussion below on tile-flavor editing.) It includes the buttons:

Box Make the component box-flavored.

Layout Make the component layout-flavored.

Field The flavor of the component will be determined recursively based on the type and value of the data that is put in that component.

Pointer The component will be drawn as a box with either a slash through it or a dot and an arc to the object representing the data that is in the component. This is only useful when the display occurs in a layout.

Ignore The component will be empty.

Empty The component will be drawn as an empty box. Note that this may differ from *Ignore* in the case where different colors are used. This should be the used in most cases.

Finally, the *Props* menu contains one button, *Options*. If there is a current selection (either a tiled component, an include component, or a layout component), then the drawing options for that component will be provided to the user. Otherwise, the options for the top level object will be provided. If there is no top-level object, the user will be asked to specify its type.

Layouts are specified by providing a list of items that should be drawn in the resultant graph. These can either be drawn as nodes or arcs. Often it is the case that adding one item to a layout, for example an element of a linked list, will cause other items to be added to the same layout, i.e. the next item in the list. This is specified in the type editor using the *Include* box at the bottom of the window. This is only effective when the current alternative is conditioned on appearing in a layout. The user can add the additional items by first clicking on the word *Include* and then selecting values from the menu at the right. Individual items can be selected by clicking on their name in the Include box or on the corresponding button on the right. Once an individual item is selected, it can be removed by clicking on the button on the right or options can be specified for it. The options include whether the item should just be added, should be added and an arc drawn from the current item to it, or should be added with an arc drawn from it to the current item.

The menu to the right of the representation area contains a list of alternative values. These values can be used to specify values for the components of various top-level objects. These include the tiles in a tiling, the items to be drawn in a layout, the source, destination and label for an arc, and the include items. This list contains a set of standard values followed by all the fields of the type being displayed. The standard values include:

This allows the user to specify an arbitrary constant text string.

<Empty>
This specifies no value.

<Computed>
This option is not implemented for data structure display.

New Field
This option is not implemented for data structure display.

<From Source>
For an item that was added because of the Include option described above, this will designate the value from which it was included.

<From Index>
For an item that is part of an array, this will designate the index number in the array for the item.

<From Self>
This designates the current value itself. This is useful, for example, in building a linked list. The default way of drawing a link element in a list within a layout is as a tiling with the various values and an pointer to the next link element. However, an initial display would not place the list element in a layout. Thus the default display for a list element should be to draw a layout with itself

as the only element.

Tile flavor representations are edited using the mouse and the representation window. The left mouse button is used for drawing new tiles and changing the size of existing ones. It should be clicked down and then dragged to form an appropriate rectangle. Rubberbanding is shown during this process. A new rectangle is drawn when the mouse starts in empty space or at a corner of an existing rectangle. A rectangle is resized by starting in the middle of one of its edges. Where this is ambiguous, the current selection will be chosen over other rectangles for resizing. Note that the editor will always force the tiling to fit in a rectangle and to have non-overlapping tiles, and will create additional tiles as necessary. The middle mouse button is used for selecting the current tile. Once a tile is selected, information about it can be specified. The drawing type can be chosen off the *Edit* menu; options for it can be chosen from the *Props* menu; and the contents of it can be chosen from the menu on the right. Note that the default type is empty if no value is chosen, box if a string value is chosen, and field (recursive selection) otherwise. The right mouse button is used to select optional characteristics of tilings, notably additional arcs and constraints. The button should be clicked down in one tile and up in another (possibly the same). It will then ask the user whether to draw or edit an arc or constraint between these two tilings.

A layout flavored representation is depicted as a box with two arrows coming out of it. When a layout is selected (either because it is the top level object or because it is the type of the currently selected tile), the user can use the menu to the right to add or remove values from the layout. Selecting an item not in the layout will add it, selecting an item in the layout will first select only it (so that options can be specified for it), and will next remove it.

A box flavored representation displays as an empty box with the desired shape in the representation panel. One value or field can be chosen for the box from the menu at the right. The Options button allows the user to specify the shape, line style, fill style, and additional properties of the given box.

An arc representation is depicted as three boxes and an arrow, one box for the source of the arc, one for the destination, and one for the label. The user can select any of these boxes by clicking in them with any mouse button. Then the value to be used for this component of the arc can be chosen from the menu at the right. The Options menu for an arc object allows the specification of the line and arrow styles as well as the source and destination location for the arc.

The type editor allows several keyboard accelerators to simulate menu buttons. These include:

Key	Menu	Button
a,A	Top	Arc
b,B	Top or Edit	Box
d,D	Top	Default
l,L	Top or Edit	Layout
o,O	Props	Options
f,F	Edit	Field
t,T	Top	Tiled

FILES

./.display_defs/*

SEE ALSO

field(1), The Brown Workstation Environment Reference Manual.

COPYRIGHT

BUGS

Too numerous to mention. Please report any found to spr@cs.brown.edu so that they can be fixed.

The data structure facility cannot update itself when the debugger is not in control.
If the program is asking for input, the display will sit there doing nothing until the debugger gets control, for example.

NAME

flowview, flowclass, flowmemb – FIELD call graph browser

SYNOPSIS

flowview [*objfile*] [-options]

DESCRIPTION

flowview is a call graph browser. It uses the FIELD cross reference database xrefdb(1) to find all functions and calls in the given system. It then creates a hierarchical display and allows the user to selectively change or view the display.

The call graph display is arranged hierarchically using the normal UNIX groups, i.e. directories, files and then functions. All functions are considered to be contained in their source files, all files are considered to be contained in their directory, and directories are viewed using the UNIX file hierarchy. This allows, for example, a whole library of routines to be grouped as one node on the display.

Directories are displayed using hexagons. Files are displayed using rectangles. Functions are displayed using circles (or ellipses). An arc between two nodes represents one or more calls from the source or a function contained in the source to the target or a function contained in the target.

The browser is able to display the whole call graph or any part of it. It can be used to selectively view the calls within a file or directory. It can be used to view the interaction between files or functions within a program. It can also be used to view a localized call graph for a given function or file. Such a localized graph only includes nodes that explicitly call or are called by the given function(s).

flowview is also tied into the rest of the FIELD environment. Generally, selecting a function node or an arc in the browser will cause an appropriate message to be sent out requesting an annotation editor (typically annotview(1)) to show the corresponding source line for the function definition or the call. Moreover, the browser looks for execution tracing messages and will highlight the currently executing function.

Three different highlighting colors are used in **flowview**. Light blue is used to indicate the currently selected node. Red is used to indicate the currently executing node. This can either be a function or a file containing the currrently executing function. Finally, green is used to show the call stack.

This system also supports user-defined groupings in addition (or in conjunction) with the groupings based on the file system. Groupings are defined by specifying a pattern string. This string is similar to a scanf-type string except that it should contain a %1s (or %1r for rest of line) for the part of the name that should be used for grouping. For example, to gather all C++ class methods based on class name, the pattern *%1s::* is used; while to gather all C++ class methods based on method name, the pattern *%s::%1s(* is used; while to gather all C++ class methods based on full method name, the pattern *%s::%1r* is used. Different patterns can be designed to adapt to different naming conventions. Predefined instances of this viewer based on specified patterns can be defined using the resource files. User defined patterns can be specified dynamically.

OPTIONS

The options, X11 resources, files, and environment variables used by these commands are the same those used by the **field** command in general and the rest of the tools of the FIELD environment. See field(1).

RESOURCE FILES

The class browser resource file can be used to define the default settings that determine how to display the class hierarchy. The definitions include:

```
FLOW:
  METHOD = gelo_method_value
  CONNMETHOD = gelo connection method
  FIXED = 0|1 -- fixed size nodes
  STANDARD = 0|1 -- standard size nodes
  CENTERED = 0|1 -- centered nodes
  DIRECTORY_SHAPE = gelo_shape
```

```
 FILE_SHAPE = gelo_shape
 FUNCTION_SHAPE = gelo_shape
 ARC_STYLE = ash_line_style
 ARROW_STYLE = arrow style
 PERT_ARROW_STYLE = arrow style
 WHITESPACE = 0 .. 100

 DISPLAY_ALL = 0|1 -- show full display
 DISPLAY_FORCE = 0|1 -- force redisplay
 DISPLAY_CALLBYS = 0|1 -- show callers
 DISPLAY_CALLEES = 0|1 -- show called rtns
 DISPLAY_FIXCUR = 0|1 -- fix size of current node
 DISPLAY_CONN = 0|1 -- insure graph is connected
 DISPLAY_LEVELS = -1, 0... -- levels of local display
 DISPLAY_ZOOM = 0... -- zoom factor for current node

 AUTO_UPDATE = . -- update on successful build
 NO_STACK = . -- don't display execution stack

 NAME_EXCLUDE = ( pattern ) -- list of patterns of names to exclude
 NAME_INCLUDE = ( pattern ) -- list of patterns of names to include

 TRACE_MODE = (ALL|DISPLAY|NONE)
 EXPAND_MODE = (ALL|FILES|NORM)
 SHORT_NAMES = .
```

The GELO method and GELO connection method are integer values that can be derived from the $PRO/include/bwe/gelo.h include file. The various shapes are GELO_SHAPE values converted to integers. The arrow styles are GELO_ARC_ARROW values converted to integers. Both of these can be found in the gelo.h include file as well. The first group of parameters are all used to control the layout algorithms used in drawing the call graph. The various DISPLAY options control the default (initial) settings to be used in setting up the call graph brower. The final two options control whether the flow graph should be automatically brought up to date after a successful system build and whether the execution stack should be highlighted or only the currently executing node. The patterns for NAME_INCLUDE and NAME_EXCLUDE are regular expressions and apply to files, functions and directories.

In addition, the resource file can be used to defined different initial instances of pattern-based groupings. These are defined as:

```
VIEWER +
 NAME = command_name
 GROUP_PATTERN = "pattern"
 GROUP_ID = id_for_grouping
 DEFAULT_NAME = group name for items not matching group pattern
```

In addition, the METHOD, CONNMETHOD, FIXED, STANDARD, CENTERED, WHITE_SPACE, DISPLAY_ALL, DISPLAY_FORCE, DISPLAY_CALLBYS, DISPLAY_CALLEES, DISPLAY_FIXCUR, DISPLAY_CONN, DISPLAY_LEVELS, DISPLAY_ZOOM, NAME_EXCLUDE, and NAME_INCLUDE properties can be specified separately for each viewer.

USAGE

The main **flowview** window is divided into two parts, a pull down menu bar and the display area for the call graph. The pull down menu consists of four different menus. The first, *Flow* is used to control the overall window. It includes:

Info Window
This button will cause a new window to be created on the current display. This window will be used by the browser to display more detailed information about the currently selected directory, file or function in textual form. For more details on this window, see below.

Groupings
This button allows the user to dynamically change groupings, either to a predefined grouping (including none), or to a new, user-defined grouping.

Restart This button will discard all information about nodes to be ignored and levels of the hierarchy to be drawn that had been specified by the user and will then redraw the current display.

Reset This button will discard all information about nodes which levels of the hierarchy should be drawn that had been specified by the user and will then redraw the current display.

Update This button will request xrefdb reload the current system and will then redraw the display. Any information about the current class or member as well as information about classes that shouldn't be displayed will be lost.

Trace This button will cause **flowview** to issue debugger commands to trace entry and exit of all functions so that execution can be fully monitored by the browser.

Trace Display
This option is similar to *Trace* above, but only causes the functions that are currently represented by nodes on the display to be traced.

Set System
This allows the user to change the system that is viewed. This is effective both for viewing another system and for interpreting and sending messages to the other tools of the FIELD environment.

Quit This removes the browser window(s) and exits the browser.

The *Selection* menu contains options that allow the user to set and modify the current selection. The buttons here include:

Set File This causes a dialog box to be put up that allows the user to enter the name of the file or directory to be selected.

Set Function
This causes a dialog box to be put up that allows the user to enter the name of the function (and its file if necessary) that should become the currently selected node.

Clear Selection
This clears the currently selected node. The display is then updated accordingly.

Set Selection
This button causes a series of one or more dialog boxes to be put up that allow the user to go through and select what should be displayed. The user is asked to select items hierarchically. For each item, there is an ignore button. Checking this button will cause the corresponding item (and all its subitems) to be omitted from further displays. The user can also request to view the items of a particular file or directory (in the next dialog box), or to make a particular item the currently selected one (by selecting on the name). If there are more items at a particular level of the hierarchy than will conveniently fit in a dialog box, then multiple dialog boxes are used and the user will be provided with the option of going to the next (or subsequent or both as appropriate) one.

Select Pattern
This button allows the user to specify regular expressions (regex(3)) to identify nodes that should be ignored or included from the call graph. The patterns can be made to apply to directories, files or functions.

The *Display* menu contains options that allow the user to customize the display. The buttons here include:

Options This button puts up a dialog box allowing the user to set any of the various display options other than those affecting the layout algorithms. The user can first select whether to show a complete graph or only a localized graph. The user can restrict the display to only nodes that are explicitly reachable from the main program using the *Only show reachable nodes* option. The *Force redisplay on selection* option will cause the display to be recomputed as necessary whenever a new selection is made. This is the default setting. The *Show calling routines* and *Show routines called by* options are used to limit a localized display to only the callers or the callees. The *Levels* option allows the setting of how many levels of callers and/or callees should be displayed in a localized graph. Finally, the options *Expand size* and *Zoom selection* can be used to highlight the current selection by insuring that it is sized so that its text is readable and emphasizing it by a given factor respectively.

Show All
This option, if selected, will cause the browser to show the complete call graph as opposed to the localized display for a given node.

Connected
This option will restrict the nodes displayed to those that are reachable from the main program via static calls.

Show Callers
This option causes the calling routines to be displayed in a localized graph.

Show Called
This option causes the called routines to be displayed in a localized graph.

Expand All
This button will cause all file and directory nodes to be expanded into the corresponding set of function nodes in the current display.

Expand Dirs
This button will cause all directory nodes to be expanded into the corresponding set of group or file nodes in the current display.

Layout This will put up a dialog box allowing the user to change or play with the layout algorithms that are used in drawing the class graph.

Finally, the *Node* menu is used to select operations specific to the currently selected node. Note that this menu is disabled if there is no selected node. The buttons here include:

Show Item
If the current selection is a file or directory, this option will cause the display to show the hierarchy contained in that node. In any case, it will cause the information window to display information about the current selection.

Show Parent
This will restrict the current display to that of the parent of the current selection.

Ignore This button cause the current item to be ignored, i.e. neither it nor any of its subnodes will be displayed.

Expand This button will expand the current selection, provided it is a file or directory node, causing it to be replaced within the current graph with all its components.

Compact
This button will compact the current selection. This causes this node and any other nodes contained within its parent to be replaced in the current graph by the node for its parent.

Show Info

This will cause a dialog box to be popped up showing additional information about the current node. It will also direct the information window to display information about the selection.

The mouse can be used within the graphics display for selecting the current node as well as related operations, all by clicking (no dragging). The operations that can be done this way are:

Click On	Button	Operation
Node	Left/Right	Make item current
	Middle	Show item information
	Shift-Left	Ignore item
	Shift-Right	Compact item
Selection	Left	Show item
	Middle	Show item information
	Right	Expand item
Arc	Left	Make to item current
	Middle	Show arc information
	Right	Make from item current

Shift here indicates that either the shift, control or meta key was pressed while the mouse click occurred. In addition, any click done without a shift/meta/control on a function node or an arc will cause a message to be sent through FIELD requesting a source display of the associated line.

In addition to mouse clicks, **cbrowse** provides a few keyboard accelerators. These include:

Key	Operation
r	Redraw display
R	Reset and redraw display
c,C	Clear current class
a,A	Toggle show all mode
u,U	Update

The information window provides more detailed information about the current class. It is run as a readonly EDT editor. When a node is made current, the information for that node will be displayed. If the information was not previously displayed then it will be added at the end of the transcript being edited. The editor will automatically scroll so that the start of the information is at the top of the window. The user can click on any line of the display. Clicking on a reference to a line will generally send a message through FIELD requesting that an annotation editor (typically annotview) display that line. Clicking on a reference to a function or a file will cause that function or file to become the currently selected item. Clicking with the right button will not select a new current item, but will only send out an appropriate message to the annotation editor.

ENVIRONMENT VARIABLES

In addition to all the standard environment variables supported by field(1), the Pascal programmer should be sure to set the variable *STDPASCAL* is names are case insensitive.

SEE ALSO

field(1), xrefdb(1), annotview(1), The Brown Workstation Environment Reference Manual.

COPYRIGHT

BUGS

Too numerous to mention. Please report any found to spr@cs.brown.edu so that they can be fixed.

NAME

formserver – FIELD configuration management server

SYNOPSIS

formserver

DESCRIPTION

formserver is the background process that serves as a clearing house for configuration management requests for the FIELD environment. It caches information about the various projects (directories) that are currently being used. It can handle requests to compile or to execute a given command for any project.

SEE ALSO

field(1), formview(1)

COPYRIGHT

BUGS

Too numerous to mention. Please report any found to spr@cs.brown.edu so that they can be fixed.

NAME

formview, transcript – FIELD make interface

SYNOPSIS

formview [*sourcedir*] [-options]

transcript [*sourcedir*] [-options]

DESCRIPTION

The make interface of FIELD provides two functions. First of all, it provides a service to other tools of the environment, handling message-based requests to compile or make a system or to execute some makefile-based command. Secondly, it provides in interactive interface whereby users can view their makefiles graphically and can view the output of the various make commands.

The **formview** command runs the full make interface, providing a visualization of the user's makefile. It operates by running the appropriate back end, either the native version of make or GNU make at this point, and interpreting the debugging output. From these is constructs an internal representation of the configuration. This representation can be viewed graphically using the facilities provided. This command also provides an editor showing a transcript of the most recent make commands.

The **transcript** command is a much simplified interface to this facility that does not allow viewing of the makefile and that only provides the transcript view. Once started, it will show a full transcript of all make-related commands. Normally, the transcript only reflects commands issued in the directory specified on the command line. If the command is run without arguments or from the FIELD control panel, than all commands issued in any directory will be reflected in the transcript.

The service handling part of this package, without any user interface, is started automatically with most of the FIELD tools. Thus, the user does not have to explicitly create a build window in order to have the *COMPILE* command in the annotation editor work.

OPTIONS

The options, X11 resources, files, and environment variables used by these commands are the same those used by the **field** command in general and the rest of the tools of the FIELD environment. See field(1).

RESOURCE FILES

The resource file for **formview** is used to define several aspects of the system. These definitions are grouped under the heading FORM.

FORM keeps an internal database consisting of nodes, links and attributes. Nodes typically represent files, links represent relationships between files, and attributes represent properties of either nodes or links. There are some built-in attributes that are known to make, and additional attributes can be defined to support additional features of the back ends such as version control. FORM can support multiple back ends. Each back end is essentially an interface to the appropriate UNIX tool. The current set of back ends includes Sun's version of make, the version of make running on the DEC 3100, and GNU's version of make. FORM also scans the output of executing make for compiler error messages that can be broadcast to other tools in the FIELD environment FORM can also support multiple configuration management back ends. In addition to running make over the directory, it will request information from the appropriate configuration manager and set the appropriate file attributes. The resource file is used to define the set of back ends, the set of configuration managers, the set of attributes, and the set of messages to be scanned for.

Backends are defined by defining their properties and their entry points:

```
BACKEND+
  NAME = name
  INIT = initialization routine
  EXEC_SCAN = scan execution routine
  EXEC_BUILD = build execution routine
  SCAN = output_scanning_routine
  PATH = filename of system to execute

DEFAULT_BACKEND = name
```

Configuration managers are defined by defining their properties and entry points as well:

```
CONFEND +
  NAME = name
  INIT = initialization routine
  EXEC_INFO = information scanning execution start
  SCAN = information scanning routine
  EXEC_CMD = command initiation routine
  DIRECTORY = subdirectory name that is used

DEFAULT_CONFEND = name
```

Attributes are defined by providing their name and their type. Attribute types are defined by giving the type class (one of Boolean, int, float, String, enum, or float), their name and any additional properites (i.e. values of an enumeration type):

```
ATTR_TYPE +
  NAME = name
  CLASS = class
  VALUES = ( list of enumeration values )

ATTR_ID +
  NAME = name
  TYPE = type name
  DEFAULT = value
  SETABLE = .
  DISPLAY = "format string"
  ORDER = ordering value
```

The interface for monitoring the result of execution of make is described by one or more monitor events:

```
MONITOR +
  TEXT = "pattern to match"
  EVENT = output event name
  FORMAT = alternative output format
```

The patterns are MSG-type patterns. The embedded arguemnt %1s refers to the filename, the argument %2d refers to the line number, the argument %3s is the message text (unless provided by FORMAT alternative), the arguments %4s, %5s and %6s are strings that are saved and can be used in the FORMAT alternative. If the monitor does not contain an EVENT value, than values are set but no message is broadcast. Values typically will be shared from one monitored event to the next.

The display interface also allows the setting of various display properties through the resource file:

```
  METHOD = gelo_method_value
  CONNMETHOD = gelo connection method
  FIXED = 0|1 -- fixed size nodes
  STANDARD = 0|1 -- standard size nodes
  CENTERED = 0|1 -- centered nodes
  ARC_STYLE = ash_line_style
  ARROW_STYLE = gelo_arc_arrow
  PERT_ARROW_STYLE = gelo_arc_arrow
  WHITESPACE = 0 .. 100
```

```
DISPLAY_ALL = 0|1 -- show full display
DISPLAY_FORCE = 0|1 -- force redisplay
DISPLAY_FIXCUR = 0|1 -- fix size of current node
DISPLAY_LEVELS = -1, 0... -- levels of local display
DISPLAY_ZOOM = 0... -- zoom factor for current node
DISPLAY_INLINKS = 0|1 -- show items depended on
DISPLAY_OUTLINKS = 0|1 -- show dependencies
PROJECT_SHAPE = gelo shape
COMMAND_SHAPE = gelo shape
SYSTEM_SHAPE = gelo shape
INTERMEDIATE_SHAPE = gelo shape
SOURCE_SHAPE = gelo_shape
CONF_SHAPE = gelo_shape
ITEM_EXCLUDE = ( pattern ) -- list of patterns of names to exclude
ITEM_INCLUDE = ( pattern ) -- list of patterns of names to include
```

In addition, automatic compilation when a file is saved can be requested by setting the flag AUTO_COMPILE = . in the resource file. Also setting CLEAR_AT_START = . will cause the transcript to be truncated at the start of each make rather than being accumulative.

In addition, formview uses the file *.formrc* in the project directory to allow for project-specific settings. These settings are used to set attributes of the given project and consist of lines containing the attribute name, an equals sign (=), and the attribute value. In particular, this feature can be used to specify an alternative make file for the project with a line
MAKEFILE=<alternative>

USAGE

A formview interface window is divided into two parts, exclusive of the window decoration provided by the window manager(s). The top of the window consists of the title bar for the pull down menus. The bulk of the window consists of graphical browser over the dependency graph. A transcript window, gotten either by running the transcript command or through the appropriate menu button in formview, simply consists of editor.

There are five pull down menus. The *Form* menu contains controlling buttons:

Transcript
This requests that transcript window be created to go along with the formview interface.

Restart This causes the interface to display the original dependency graph. It resets the set of ignored items to the appropriate defaults.

Reset This will remove the current selections and display the full current dependency graph.

Update This will cause the interface to rerun the backend to update its dependency and attribute information.

Set Project
This button allows the user to set the project or directory that this viewer is looking at.

Quit This button closes the window and terminates the interface.

The *Make* menu contains buttons to run the back end. These include:

Make Current
This will invoke the back end to make the currently selected item.

Make Default
This will invokde the back end to make the default item in the appropriate directory.
This will either be the directory of the currently selected item, the directory that formview was invoked with, or the current working directory.

Make ... This will allow the user to interactively decide what to make.

The *Configure* menu contains buttons to run the configuration manager back end. These include: This will update configuration information about the selected file(s) or the whole project. This will check in the selected file(s) or the whole project. This will check in files selected by the user in a dialog box. This will check out the selected file(s) or the whole project. This will check out files selected by the user in a dialog box. This will remove unchanged locked versions of the selected file(s) or the whole project. This will remove unchanged locked versions of files selected by the user in a dialog box.

The Selection menu is used to select items for the graphical dependency display:

Set Selection
: This allows the user to enter a new selection textually.

Clear Selection
: This clears the current selections.

Select Item
: This puts up a dialog box (or a sequence thereof) that allows the user to pick the new selection and to selectively ignore of view different items.

Item Patterns
: This allows the user to specify regular expression patterns describing items (eg. files) that should be included or excluded from the display.

Finally the *Display* menu contains options to control the graphical presentation of the make dependencies:

Options This puts up a dialog box to allow the user to set all the various display options.

Show All
: Formview's display operates in one of two modes. Either it always shows the whole dependency graph, or it just shows the graph that is local to the currently selected node. This button toggles between these modes.

Show Dependencies
: When showing only the graph local to the current node, formview can selectively show either the dependencies of the node, the items it is depended on by or both. This button toggles the display of dependencies.

Show Uses
: As above, this button toggles the display of items depended on by the current selection.

Layout This button provides a dialog box to allow the user to vary the layout heuristics that are used in the display.

The mouse can be used within the graphics display for selecting the current item as well as related operations, all by clicking (no dragging). The operations that can be done this way are:

Click On	Button	Operation
Item	Left	Select, expand if necessary
	Middle	Show item information
	Right	Select
	Meta-Left	Ignore item
	Shift-Left	Add/remove item form selection set
Selection	Middle	Show item information
	Right	Cancel selection

Shift here indicates that either the shift, control or meta key was pressed while the mouse click occurred.

In addition to mouse clicks, **formview** provides a few keyboard accelerators. These include:

The user can also click on the transcript window. Clicking here on an error message will send an XREF message out, causing an appropriate editor (such as annotview) to go to the corresponding line number.

Key	Operation
r	Redraw display
R	Reset and redraw display
c,C	Clear current selections
a,A	Toggle show all mode

SEE ALSO

field(1), make(1), The Brown Workstation Environment Reference Manual.

COPYRIGHT

BUGS

Too numerous to mention. Please report any found to spr@cs.brown.edu so that they can be fixed. This interface needs a significant amount of work to become more relevant and more easily used. One possiblity is to use a browser interface such as with cbrowse(1) or flowview(1). Another possibility would be to replace it with a reasonable interface to some public domain configuration management system that is more powerful and comprehensive than make.

NAME

freex, freeserver, freerem – FIELD remote execution facilities

SYNOPSIS

freex [-sh] [-localsh] [-exec] [-localexec] command ...

freeserver

freerem host port hostname userid maxuser maxload

DESCRIPTION

freex is a remote execution startup runs it on a free machine somewhere in the network. To do this, it requires that **freeserver** is run. This process is automatically started if it is not currently running (i.e. the user should never have to run it). The command **freerem** is then run on the remote machine by freeserver (i.e. the user should never need to use this command either).

Freeserver takes a file (˜/.freerc) that describes the hosts in the network and uses it to find idle remote hosts to run on. Freex is now used by the version of gnumake that is distributed with FIELD if the user sets REMOTE (either as an environment variable or in the Makefile). REMOTE should be set to the number of remote jobs that should be run in parallel. The variable NO_REMOTE can be set to a space-separated list of commands that should be run locally. If it is not set, reasonable defaults are provided.

USAGE

The actions of **freeserver** and hence **freex** are determined by the freerc files that are found. FREE will use the system file installed in the appropriate FIELD rundata directory if there is one. It will also use the .freerc file in the user's home directory. This file consists of lines, one entry per host. The file must include any machines to be used remotely as well as all machines from which remote execution is to be attempted.

The file contains one line per host. Each line has the format:

```
    hostname architecture filesystem freerem_command users load
```

The hostname is the name of the host (suitable for rsh). The architecture and filesystem allow free to be used in a heterogeneous network. They are used by freeserver to compare the host initiating the commands with suitable remote hosts. The architecture describes the hardware type (i.e. sun3, sun4, mips), while the filesystem describes nodes that share a common NFS file system (i.e. pathnames will match). (free will remove an initial /tmp_mnt from a pathname, but will assume that the rest of the pathname is valid on both machines.) The freerem_command is a full pathname for the remote execution package freerem on the remote machine. If it does not start with a '/', it is assumed to be in the FIELD binary directory accessible in the same way on the host and on the remote machines. The users field indicates the maximum number of users (the actualy number of users on the remote machine must be less than this value or the machine will not be considered idle). A users value of zero will cause the machine not to be used. Finally, the load is a real number which the load (given by uptime for the past 1 minute) must be less than for the machine to be considered idle.

OPTIONS

-sh	Use /bin/sh instead of the user's default shell.
-localsh	Use /bin/sh and run the job locally.
-exec	Execute the job directly (rather than going through the user's shell).
-localexec	Execute the job directly on the local machine.

ENVIRONMENT VARIABLES

FIELD_TMP

This indicates the temporary directory to be used. It defaults to /usr/tmp. The directory cannot be a tmpfs type file system on a Sun.

FREE_RSH
: This contains the command name to start up freerem. It defaults to /usr/ucb/rsh.

FREE_LOCAL
: If this is set, then jobs will be run on the local machine if no remote machines are available rather than being queued.

FREE_DEBUG
: If this is set, information about what FREE is doing is printed.

FILES

˜/.freerc
$PRO/lib/field/rundata/$ARCH/freerc
$PRO/field/rundata/$ARCH/freerc

SEE ALSO

field(1), xref(1), flowview(1), cbrowse(1), cpp(1)

COPYRIGHT

BUGS

Too numerous to mention. Please report any found to spr@cs.brown.edu so that they can be fixed.

NAME

msgserver – FIELD message server

SYNOPSIS

msgserver [-D] [-force] [-msg *file*]

DESCRIPTION

msgserver is the background process that acts as the central message facility of the FIELD environment. If it is not already running when a FIELD tool (other than ddt(1) without the -f option) starts up, then it will be run automatically. It includes several locks and checks so that only one copy of the server is running at a time. It can also be run manually if desired.

msgserver operates by creating a tcp/ip socket. It writes its hostname and the port number for this socket in a file and locks this file using lockf(3). This file is generally the file /usr/tmp/msg.*hostname*.*userid* where the *hostname* is the name of the host machine and the *userid* is the user's login name. Another file can be specified using the -msg option. Whenever a FIELD tool starts running, it looks for this file and, if found, checks to make sure it is locked. If it is, then it reads the hostname and port from the file and connects to the corresponding message server. If the file doesn't exist or if there is a problem connecting, then it will run a new message server on the given file.

The message serving utility will work across machines if the files are accessible network wide and the lockf function works. Multiple messages servers can run on the same machine if they have different file names.

OPTIONS

-D This runs the message server in debugging mode. If it can, it will create a file msg.trace in the current directory and will record a transcript of all messages received and sent in this file. If it doesn't have permission to write to this file, it will use stderr.

-force This causes the message server to start up regardless of the lock state of the lock file. This is useful on machines where the lockf(3) utility is unreliable, as it is on our Decstations currently.

-msg *file*
The given file is used as the lock file for this server.

FILES

/tmp/msg.*hostname*.*userid*

SEE ALSO

field(1)

COPYRIGHT

BUGS

Too numerous to mention. Please report any found to spr@cs.brown.edu so that they can be fixed.

NAME

userio – FIELD user input/output window

SYNOPSIS

userio [-options]

DESCRIPTION

This command provides a simple EDT editor window tied to a pseudo tty (pty). It uses the FIELD message service to inform all active ddt(1) debuggers that the default user input and output (stdin and stdout) should be directed to this pty. The effect is that program input and output goes to this window rather than being intermixed with the debugger transcript in the debugger window.

userio places a header and trailer in the output stream whenever the program starts or stops execution. Once a **userio** window is created, it will be used by all debuggers (there is no way currently to restrict it to a single system) starting with the next time a program is run.

OPTIONS

The options, X11 resources, files, and environment variables used by these commands are the same those used by the **field** command in general and the rest of the tools of the FIELD environment. See field(1).

SEE ALSO

field(1), dbg(1), ddt(1), The Brown Workstation Environment Reference Manual.

COPYRIGHT

BUGS

Too numerous to mention. Please report any found to spr@cs.brown.edu so that they can be fixed.

The terminal simulation capabilities of EDT are primitive and this package probably will not handle programs that depend on intelligent terminal output. It should be possible to put up an xterm that does the same thing rather than running EDT. Alternatively, the pty interface to EDT might be fixed sometime.

NAME

viewevent, viewstack, viewtrace − FIELD viewers

SYNOPSIS

viewevent [*objfile*] [-options]

viewstack [*objfile*] [-options]

viewtrace [*objfile*] [-options]

DESCRIPTION

These commands provide simple viewers onto the FIELD message facility. They each utilize a text editor to show the current status of different classes of messages. In addition, they can provide the ability to issue commands.

The command **viewevent** provides a listing of current debugger events. This would include breakpoints, tracepoints, watchpoints, interesting events, displays, updates, etc. It currently provides no interface for adding or removing events however. **viewstack** provides a display of the current execution stack. This display is updated whenever the debugger gets control of program execution. **viewtrace** provides a display of the most recent values of watched or displayed variables or expressions. It also provides a facility for requesting such displays. When it is started, it asks the user to specify a variable to trace. This request can be ignored if desired.

OPTIONS

The options, X11 resources, files, and environment variables used by these commands are the same those used by the **field** command in general and the rest of the tools of the FIELD environment. See field(1).

RESOURCE FILES

These three commands are actually instances of the same package. The resource file for this package is used to define the particular commands by defining the message patterns that each monitors. Each message pattern can be used to add or remove a displayed line of text. The lines are ordered by a key found in the messages. The resource file heading for these commands is *VIEW*.

Messages are grouped together in classes. Each class defines a potential display line by giving the message that causes that line to be displayed, the message that causes it to be removed, and the format of the display. The definition is:

```
CLASS +
  NAME = name
  DISPLAY_MSG = pattern
  DELETE_MSG = pattern
  END_MSG = pattern
  DISPLAY_FMT = format
```

The NAME field identifies the class. The various MSG fields are used to define message patterns. These are registered directly with the message server. They can contain up to six arguments and must contain at least one. The first argument (%1s) must be a string argument. It is used as the key, i.e. it associates the incoming message with a display line and provides a sort order. Note that while it is handled here as a string, it can still be a number for sorting purposes. A message matching the DISPLAY_MSG pattern will cause a display line to be created (or replaced). The format of this display line is given by the DISPLAY_FMT line. This can contain references such as %3s to the arguments in the message pattern. A message matching the pattern DELETE_MSG will cause the corresponding message to be removed. A message matching the pattern END_MSG will cause any displays whose key is greater than or equal to that of the given message to be removed.

The different viewers are defined in the resource file by defining the classes of messages that they accept and any additional properties that are relevant. The definition here is:

```
TYPE +
 NAME = name
 USE = ( message class list )
 NUMERIC_KEY = .
 OPEN_MSG = message
 CLEAR_MSG = pattern
 OPTIONS = rtn
 OPTIONS_NAME = btn_name
 REMOVE = rtn
 REMOVE_NAME = btn_name
 ASK_FIRST = .
```

The NAME field identifies the viewer type. The USE field contains the classes of messages that this viewer handles. The NUMERIC_KEY option indicates that the key should be interpreted as a floating point number for sorting and matching purposes. The OPEN_MSG is a message that is sent when the viewer is created. It is generally a command that will generate messages for an initial display. It can have one %s argument in it which will be filled in with the relevant system name. The CLEAR_MSG is a message pattern with one argument, a key. If a message is received matching this pattern, then all current displays are removed. The OPTIONS line specifies a C function that will be tied to a button on the viewer pull down menu. This will either be the *Option* button or will be the name specified by OPTIONS_NAME. Similarly, the REMOVE line specifies the function and the REMOVE_NAME the button name for a button that will allow the user to remove displays selectively. Finally, the ASK_FIRST option, if specified, will cause the options buttons to be invoked implicitly when the viewer is started.

USAGE

A viewer consists of a pull-down menu window and a readonly EDT editor to contain the message displays. The menu bar contains a *View* menu with options relevant to this particular viewer, and a series of menus for the EDT editor. The *View* menu contains the following buttons:

Set System
This lets the user specify what system (object file) messages should be viewed for. The defualt is all systems.

Options (**viewstack** only)
This puts up a dialog box that allows the user to specify whether the stack display should include local variables and or global variables. It also allows the user to limit the amount of the stack that is shown to some number of frames at the top of the stack and some number of frames at the bottom.

Variable to trace (**viewtrace** only)
This puts up a dialog box that allows the user to generate a watch or display event for a given variable or expression. The user can give an expression or variable name, restrict the location of the trace to a given function or line (which is more efficient), and distinguish between a display event (only updated at points where the debugger gets control) or a watch event.

Remove trace (**viewtrace** only)
This puts up a dialog box that allows the user to remove all watch or display events for any of the variables or expressions currently being displayed.

Quit This terminates the viewer.

SEE ALSO

field(1), ddt(1), The Brown Workstation Environment Reference Manual.

COPYRIGHT

BUGS

Too numerous to mention. Please report any found to spr@cs.brown.edu so that they can be fixed.

NAME

xprof – FIELD profiling interface

SYNOPSIS

xprof [*binary* [*monitor_data*]] [-options]

DESCRIPTION

The profiling interface of FIELD, **xprof**, provides a consistent graphical interface to a variety of profiling packages including prof (1), gprof (1), pixie on Decstations, the FIELD-modified version of gprof (fieldgprof), and our instruction-level sun profiler iprof.

xprof takes a binary file as its argument. It presumes that the program has already been run and that profiling data has been gathered. It automatically determines what profiler is being used, runs that profiler, and provides an interactive graphical presentation of the data. In the future it should be possible to have **xprof** actually run the program (either inside or outside of field), and to provide an incremental profile while the program is running.

OPTIONS

The options, X11 resources, files, and environment variables used by these commands are the same those used by the **field** command in general and the rest of the tools of the FIELD environment. See field(1).

RESOURCE FILES

The profiling interface of FIELD allows the definition of display properties and the different profilers that are available. These are all gathered under the heading XPROF. The drawing property definitions include:

```
 OLD_COLOR = color            -- color for default (and old) display
 OLD_FILL = fill-number       -- fill style for this
 NEW_COLOR = color            -- color for incremental updates
 NEW_FILL = fill-number       -- fill style for this
 OUTLINE_COLOR = color        -- color for outlining bars in bar graph
 OUTLINE_FILL = fill-number   -- fill style for this
 MIN_FONT_SIZE = size         -- smallest font size to use for labels
```

The different backends that are available to xprof are defined both with internal code and with a definition in the resource file. The resource definition includes:

```
BACKEND +
 NAME = name
 INIT = initialization_routine
 DATA_FCT = data_gathering routine
 SCAN_FCT = output line scanning routine
 INFO_FCT = information output routine
 PATH = "pathname to exec"
 KEYSYM = name               -- symbol to look for in binary
 AUXFILE = name              -- filename to look for
 RESOURCE = "resource name"
 UPDATE_REQUEST = .          -- can do update on request
 UPDATE_TIMED = .            -- can do update based on time
 UPDATE_CONTINUOUS = .       -- can do continuous updates
```

The code identified by the routines is currently in one source file for all the current profilers (prof,gprof,fieldgprof,iprof,pixie). It should be reasonably straightforward to add other profilers.

USAGE

The **xprof** window is divided into four parts. At the top is the menu bar containing the various pull-down menus. Below this, on the left half, is the tag window containing the names of the files, functions, or lines that are being profiled and the value. On the right is a bar graph display of the current profile data. Finally, there is a scroll bar all the way on the right for manipulating the label and bar graph windows.

There are three pull down menus. The *Profile* menu contains the basic operations for manipulating the window. The buttons here include:

Info Win
This button puts up an editor window to contain additional information about the currently selected profiling item.

Set System
This allows a different system to be profiled.

Restart This reinitializes the display with the current profiling data.

Reset This resets the display but does not reinitialize the various display options.

Update This causes the profiling data to be recomputed by rerunning the back end. The display is reinitialized.

Quit This terminates the make interface.

The second menu, *Selection*, is used to set or clear the focus for a more detailed look a the profile information. It includes the buttons:

Clear Selection
Clear the current selection.

The third menu contains display options. The buttons here include:

Options This button pops up a dialog box of display options. The current display options include the sort order (by value or by address as well as ordered within their parent, ie. lines ordered within functions which are sorted); bar graph properties -- how to size the bar graph display based on either the total value or the maximum value; display options including whether to make the display scroll or to force it to fit, whether to include items that have zero counts, whether to show actual values or percentages, and whether to put labels on each item or only on groups of items; and finally what items to display, lines, functions, or files.

Full This button toggles between a scrollable and a full display.

Show Zeros
This button toggles whether items with zero counts are displayed.

Show Percent
This button toggles whether the numbers shown in the label window are the actual values or are the percentages of the total.

Files This button indicates that files should be displayed. It implies no sorting by group.

Functions
This button indicates that functions should be displayed. It implies no sorting by group.

Lines This button indicates that lines should be displayed. It implies sorting by group.

The mouse can be used within the label and the bar graph displays. Any click will send an appropriate FIELD message (XREF) that can be used by the annotation editor to display the corresponding line (if **xprof** has enough information to determine the line). Clicking with the middle button in either view will put up a dialog box showing more detailed information about the clicked-on location. Finally, clicking with the left button a label will cause that label to be the current selection, providing more detailed information about it only.

SEE ALSO

field(1), prof(1), gprof(1), The Brown Workstation Environment Reference Manual.

COPYRIGHT

BUGS

Too numerous to mention. Please report any found to spr@cs.brown.edu so that they can be fixed.

NAME

xref, xrefview – FIELD cross-referencer interface

SYNOPSIS

xref [*file*] [-options]

xrefview [*file*] [-options]

DESCRIPTION

xref is a mouse oriented interface to the cross-referencing facilities of the FIELD programming environment. FIELD uses the xrefdb(1) cross-reference database to gather and provide information about the source files of a given system. It works for C, Pascal and for C++ (AT&T version 2.0). **xref** allows the user to request information from this database using dialog boxes and uses an EDT editor to display the result of the queries.

xref can be run with either the name of a object file or a single source file. If it is run with an object file then all the source files that were used to build that object file (provided they were compiled with debugging information) will be cross referenced. If it is run with a single source file, then only that file will be used.

xrefview is a paired FIELD tool. It consists of a window that is divided in half, with the upper half running the **xref** interface and the lower half running the **annotview** interface.

OPTIONS

The options, X11 resources, files, and environment variables used by these commands are the same those used by the **field** command in general and the rest of the tools of the FIELD environment. See field(1).

Two particular environment variables are needed to have the cross referencer work correctly when information about source directories is not available or when preprocessor options are given. The environment variable *INCLUDE_PATH* consists of a colon-separated list of directories in which include files should be looked for. This corresponds to a set of -I options on a compilation line. The environment variable *XRDB_FLAGS* consists of a string of -D and -U options that would be passed to the C preprocessor cpp. Xrefdb will run the C preprocessor at times and, for accuracy, whatever options are given when the system is built should be given to the cross referencer.

When using **xref** with C++ 2.0 additional care must be taken. Because of the way that C++ works, no information about source directories is included in the object file. In this case, either xref should be run in the source directory or all actual source directories should be included in the *INCLUDE_PATH* variable. In addition, the environment variable *CC_COMMAND* should be set to the name of the CC command if it not CC.

RESOURCE FILES

The AUXD resource file facility of BWE is used by XREF to define the queries that it will support. The description is provided in terms of a set of QUERY blocks, each of which contains a set of FIELD blocks. The FIELD descriptors include:

```
FIELD+
  NAME = <field name>
  WIDTH = field width
  OWIDTH = field width for output
  KEY = "R.field"     -- for output
  USE = "R.field ~= ‘%s" -- for query
  DISPLAY_FMT = <format string for output>
  INIT = Selection | File | Function | <initial value>
  COLUMN = column number for input menu
  QUERY_SHOW = .      -- ask user whether to output this field
  MUTLIPLE = <# items/output line>
  NOMATCH = .         -- imply no match for output continuation
  CONCAT = .          -- concatenate output to previous field
```

```
MUST = .            -- value must be given in input
BOTH = .            -- use for both input and output
TYPE = Integer | String | File | Boolean | Enum | Mutliple
ENUMS = ( val1 val2 ... valK )      -- for Enum or Multiple
```

These are contained in QUERY descriptors that include:

```
NAME = <name of query>
NOSORT = .          -- indicates output shouldn't be sorted
HEADER = .          -- indicates output should include header
```

in addition to the set of FIELD descriptors.

USAGE

A **xref** window consists of a pull-down menu bar and a readonly EDT editor to display the output of cross reference requests. The pull down menu consists of two local menus, *Xref* and *Query* and additional menus provided by the EDT editor.

The buttons on the *Xref* menu include:

Set System
This button allows the user to change the system being cross referenced in this window.

Reload System
This button requests that the cross reference database reload the system being viewed, thereby rescanning any source files that have changed. This would be needed if source files change outside of the FIELD environment. Any active cross-referencer is automatically reloaded after a successful build within the FIELD environment.

Quit This button causes the cross reference interface to exit.

The buttons on the *Query* menu handle cross reference queries. They operate by popping up a dialog box that allows the user to build a query, and then processing the query and putting the result up in the editor window. Leaving entries in the dialog box empty causes the query to not specify that particular value. Where strings are specified, they are generally pattern matched using regex(1) matching rules. The buttons here include:

References
This initiates a query for a reference to a given name. The dialog box allows the specification of the name, the file and the line number of the reference as well as whether the request should be limited to assignments or not. The name field is filled in with the current editor selection. This is the last selection made in and EDT editor. Output consists of the name, the file, and a list of line numbers. An asterisk after the line number indicates an assignment.

Declarations
This initiates a query for a definition of a given name. The dialog box allows the specification of the name, the file, the line number and the type as well as the class of definition. The name is filled in with the last EDT editor selection. The output consists of the name, the file, the line number, the declaration class and the data type.

Calls This initiates a query about function calls. The dialog box allows the specification of the function being called, the function it is called from, and the file and line number of the call. The output lines consist of the function being called, the function doing the calling, the filename and the line number.

Functions
This initiates a query about a function definiton. The dialog box allows the specification of the function name, the file and the line number. The function name is filled in with the current editor selection. The output lines consist of the function name, the file and line number of its definition, and the list of parameter names.

The editor portion of an **xref** window can also be used as part of the FIELD environment. The user can use the mouse to click anywhere on the query output display. The cross-reference interface will then send out a message through the FIELD message server requesting that any appropriate annotation editor display the file and line corresponding to the query output that was clicked on. If **xrefview** is run, then the annotation editor in the bottom window pane will change its display accordingly. Note that clicking will generally move the current file position within the editor. If this is not desired, the user should shift-click or meta-click. Where multiple line numbers occur on a given output line, the user can click on a particular line number. This should be done, however, with either the left or the middle mouse button.

SEE ALSO

xrefdb(1), annotview(1), field(1), The Brown Workstation Environment Reference Manual.

COPYRIGHT

BUGS

Too numerous to mention. Please report any found to spr@cs.brown.edu so that they can be fixed.

NAME

xrefdb – FIELD cross reference server

SYNOPSIS

xrefdb [-b] [-l] [-C] [-Z] [-I *directory*] [*system*] [-F*flags*]

DESCRIPTION

xrefdb is a UNIX cross reference utility. It builds a relational database by scanning the source programs for a given binary system, a given directory structure, or a single source file. It provides a relational algebra for accessing this information. The interface provided by **xrefdb** is aimed primarily at other programs. This is the utility run, for example, by the xref(1), flowview(1), and cbrowse(1) tools of the FIELD environment.

This package works by running a scanning program over the source. Scanners currently exist for C, C++ and Pascal. The result of the scanners is read in and compiled into the database. **xrefdb** is capable of incrementally updating a system or directory database so that it just scans files that have changed since the last time the database was used. The process of scanning a large system can be time consuming. The initial scan for the FIELD system (a total of about 100,000 lines of code and a 6 megabyte database) takes slightly under 30 minutes on a Sun4. Once the initial scan is complete, loading the database is much faster (a minute or two).

OPTIONS

-b This option is used to indicate that **xrefdb** is being run in batch mode. In this case no prompts are given and all messages sent back are terminated by a line containing only a special character sequence, currently ten asterisks.

-l This option can be given to force **xrefdb** to rescan the database and not to use a saved database. This is helpful when the saved database, for any of a variety of reasons, becomes corrupt.

-C This option forces the system to be treated as a C++ (2.0) program and scanned accordingly.

-Z This option causes the database to be saved in compressed form (compress(1)). If the database is originally in compressed form, then it will be saved in that form regardless of the setting of this flag.

-I *directory*
Multiple directories can be specified using multiple instances of this option. These are directories in which to look for include files. The list here should include all the directories (in the same order) that are included in compiling the components of the system. These can also be specified using the INCLUDE_PATH environment variable.

-F*flags* This options allows the specification of the compiler options that are to be used for cross-referencing (such as -D and -U).

system The system given can be a single source file, a directory or a system file. If it is a directory, then all source files (.c, .C or .p) in that directory are scanned. If it is a system, then the binary file itself is read and all source files that were compiled with debugging turned on are scanned. In either of these cases the resultant database is cached. If it is a single source file, then that file alone is scanned and no database is saved.

USAGE

xrefdb offers a very simple input language. This language accepts a small set of general commands and a query command. All commands are terminated by a semicolon. The general commands include:

Syntax	Description
# @ *system* ;	Load or reload the specified system
# + *directory* ;	Add the directory to the include list
# + ;	Clear the include list
# ? ;	Provide a list of binarys included

A query command to **xrefdb** consists of a list of items to return and a selection expression that qualifies which items should be returned. Items are identified by a relation id and a field name, separated (without spaces) by a period. The relation id can either be the name of the relation or can be any alphanumeric string begining with the key letter identifying the relation. This allows the same relation to be used multiple times in a query in different contexts.

The *Scope* relation is used to describe scopes:

Relation	Key	Field	Type
Scope	S	class	SCOPE_CLASS
		id	SCOPE
		start	int
		end	int
		file	FILE
		inside	SCOPE

The *class* field can be one of *EXTERN*, *INTERN*, *ARGS*, or *SUE*. It denotes the type of scope. The *start* and *end* fields denote the start and end line numbers in the given file. The *inside* field is used to reflect scope nesting.

The *Ref* relation contains information about references to names. Each reference includes the file, line and function (or *TOP* if it is outside of all functions), and a flag indicating whether the reference denotes an assignment:

Relation	Key	Field	Type
Ref	R	name	NAME
		file	FILE
		line	int
		assign	Boolean
		function	NAME

The *Decl* relation contains information about declarations.

Relation	Key	Field	Type
Decl	D	name	NAME
		scope	SCOPE
		file	FILE
		type	TYPE
		class	DECL_CLASS
		line	int
		id	DECL
		function	NAME

The *scope* field identifies the current scope at the point of declaration.
The *file, line* and *function* field identify the point of declaration. The *class* field denotes what is being declared. It can be one of *STATIC*, *EXTERN*, *AUTO*, *REGISTER*, *TYPEDEF*, *EXTDEF*, *PARAM*, *FIELD*, *EFUNCTION*, *SFUNCTION*, *STRUCTID*, *UNIONID*, *ENUMID*, *CONST*, *CLASSID*, or *MACRO*.

The *Call* relation contains information about function calls. It includes:

Relation	Key	Field	Type
Call	C	from	NAME
		call	NAME
		file	FILE
		line	int

The *Fct* relation contains information about function definitions. The fields here are:

Relation	Key	Field	Type
Fct	F	name	NAME
		file	FILE
		line	int
		scope	SCOPE
		numarg	int
		args	char *

The *args* parameter is a comma-separated list of argument names.

The *File* relation describes files:

Relation	Key	Field	Type
file	f	name	NAME
		tail	NAME
		id	FILE
		usedby	FILE

The *name* field denotes the full pathname. The *tail* field denotes the actual filename with directory information removed. The *usedby* field reflects include dependency information.

The *Hier* relation contains information about C++ class hierarchies. It includes:

Relation	Key	Field	Type
Hier	H	class	NAME
		parent	NAME
		file	FILE
		line	int
		public	Boolean
		virtual	Boolean
		friend	Boolean

The *Memb* relation contains information about C++ class members. For each member it includes:

Relation	Key	Field	Type
Memb	M	class	NAME
		member	NAME
		file	FILE
		line	int
		protect	PROT_CLASS
		isdata	Boolean
		inline	Boolean
		friend	Boolean
		virtual	Boolean
		static	Boolean
		pure	Boolean
		const	Boolean

The *protect* field denotes the protection type for this member. It can be one of *PUBLIC*, *PRIVATE*, or *PROTECTED*. Additional information about C++ class members is found in the *mDef* relation. This information is actually computed by **xrefdb** from the other relations and is not found during scanning. It describes the declaration associated with this member and includes:

Relation	Key	Field	Type
mDef	m	class	NAME
		member	NAME
		file	FILE
		line	int
		type	TYPE
		name	NAME

The *name* field here corresponds to the fully expanded, C++ demangled name.

The syntax for queries in **xrefdb** is as follows:

```
query           ::=   output_list selectors ;

output_list     ::=   ( output_fields )
                 |    (* output_fields )

output_fields   ::=   field
                 |    output_fields , field

selectors       ::=
                 |    selector_expr

selector_expr   ::=   ( selector_expr )
                 |    selector
                 |    selector_expr & selector_expr
                 |    selector_expr | selector_expr

selector        ::=   field == expr
                 |    field != expr
                 |    field <= expr
                 |    field < expr
                 |    field > expr
                 |    field >= expr
                 |    field ~= expr

expr            ::=   field
                 |    ‘ integer
                 |    ‘ float
                 |    ‘ " string "
                 |    ‘ ’ char ’
                 |    ‘ name
                 |    @text@
```

The *field* terminal is an identifier of the form R.f where R is the relation name and f is the field name as described above. If the output list starts with an asterisk, then the output will not be sorted; otherwise it is sorted on the fields in the order they are given and duplicate items are eliminated. The Boolean operators **&** and **|** are left-associative and of equal priority.

Selectors depend on the field type of their left-hand operand. All types can be checked for equality or inequality. Field types that are integer, string or name can have be tested for less than or greater than as well. Field types that are NAME or string can be tested using regular expression patterns with the operator **~=**. These are defined in regex(3). Note that complex patterns and names should be specified using the @...@ syntax. Also note that other constants are expected to have a backquote in front of them.

xrefdb needs to know how to use cc (CC,pc) on any C (C++,Pascal) source files in order to scan them. This means that it needs to know the set of include files that are used and any options that need to be specified. These can be specified on the command line (via the -I and -F options). This is not practical, however, when **xrefdb** is run from another program. They can thus also be specified via the environment variables INCLUDE_PATH and XRDB_FLAGS respectively. Finally, they can be set on a system-by-system basis using a control file. This file is called *stored in the same directory as the system. This file consists of command lines. These lines currently have the form INCLUDE <name>, the form FLAGS <flags>, the form CPLUSPLUS indicating that all source files in the directory should be assumed to be C++ sources, or the form COMPRESS indicating that the database should be saved in compressed form. Other commands might be added in the future.*

ENVIRONMENT VARIABLES

CPLUS20
: If this is set, then all .c source files are assumed to be C++ sources.

INCLUDE_PATH
: This is a colon-separated list of directories that should be searched for include files.

CC_COMMAND
: This is the command that should be used to run the C++ compiler. It defaults to CC.

XRDB_FLAGS
: This is a string containing -D and -U options to the C preprocessor. Any options that are normally specified during compilation should be defined here since the various scanners run the C preprocessor.

FILT_COMMAND
: This contains the command to demangle names (default $PRO/bin/ddtfilter).

XREF_CFRONT
: This allows an alternate cfront to be provided. If this is given, the +X option is also passed to this cfront.

XREF__CCARGS
: This allows the user to specify a completely different set of arguments to CC_COMMAND for xrefing. The default args are +e1 -c -F +d +a1.

XREF_INLINE
: If this variable is set, then inlines will be expanded before cross-referencing.

XREF_NOUNLINK
: If this variable is set, the g++ .gxref files will not be unlinked. This uses a lot more disk space, but avoids the necessity of having to recompile a file before its cross reference is valid.

XREF_NOKEEPDB
: If this variable is set, then no database is saved and any old one will be removed after it is read.

XREF_NOUPDATE
: If this variable is set, then the database is not automatically brought up to date the first time it is loaded. A subsequent reload will bring it up to date, however.

XREF_IGNORE
: A colon separate list of regular expression patterns specifying files to ignore.

FILES

FIELD is designed to be installed in subdirectories of a given host directory. At Brown, this is either /pro or /cs depending on the version of FIELD that is being used. In other installations, it may be an arbitrary directory. We will designate it $PRO. The architecture name (via the arch command on suns) is used where multiple systems must be supported from a common hierarchy. This is designated $ARCH.

.*.xref -- stored databases
.*.xrefrc -- resource files
$PRO/bin/field/$ARCH/xrefscan -- C scanner
$PRO/bin/field/$ARCH/xrefcpscan -- C++ scanner
$PRO/bin/field/$ARCH/xrefpscan -- Pascal scanner

SEE ALSO

field(1), xref(1), flowview(1), cbrowse(1), cpp(1)

COPYRIGHT

BUGS

Too numerous to mention. Please report any found to spr@cs.brown.edu so that they can be fixed.

NAME

xrefserver – FIELD cross reference server

SYNOPSIS

xrefserver

DESCRIPTION

xrefserver is the background process that serves as a clearing house for cross reference database requests for the FIELD programming environment. It handles the various messages requesting databases to be loaded or reloaded and requesting information from a cross reference database (via a QUERY message). It can run multiple xrefdb databases simultaneously.

SEE ALSO

field(1), xrefdb(1)

COPYRIGHT

BUGS

Too numerous to mention. Please report any found to spr@cs.brown.edu so that they can be fixed.