



Vx960™

User's Guide to VxGDB
A Source-Level Debugger
for Vx960

Release 3.2

Intel Corporation

Copyright © Intel Corporation 1991
Portions Copyright © Wind River Systems, Inc. Reproduced with permission.

ALL RIGHTS RESERVED.

No part of this publication may be reproduced in any form, by photocopy, microfilm, retrieval system, or by any other means now known or hereafter invented without the prior written permission of Intel Corporation.

Intel Corporation makes no warranty for use of its products and assumes no responsibility for any errors which may appear in this document nor does it make a commitment to update the information contained herein.

Intel retains the right to make changes to these specifications at any time, without notice.

Contact your local sales office to obtain the latest specifications before placing your order.

The following are trademarks of Intel Corporation and may only be used to identify Intel products:
i960, Intel, Vx960

COMPANY AND PRODUCT NAMES USED IN THIS DOCUMENT ARE TRADEMARKS OR REGISTERED TRADEMARKS OF THEIR RESPECTIVE CORPORATIONS.

Additional copies of this manual or other Intel literature may be obtained from:

Intel Corporation
Literature Sales
P.O. Box 7641
Mt. Prospect, IL 60056-7641

Order No. 517914

Using VxGDB with Vx960

Contents

1.	Introduction	3
1.1	Overview	3
1.2	Documentation	6
2.	Getting Started with VxGDB	6
2.1	VxGDB Configuration	6
2.2	Invoking VxGDB	8
3.	VxGDB Graphical User-interface	8
3.1	Windows	9
3.2	Window Operations	11
3.3	Command Buttons	11
3.4	Displaying C Data Structures	14
4.	Debugging with VxGDB	16
4.1	Specifying Source Directories	16
4.2	Connecting to a Vx960 Target	16

4.3	Compiling an Application Module	17
4.4	Downloading to a Vx960 Target	17
4.5	Starting to Debug	18
4.6	Using VxGDB with the Vx960 Shell Debugger	19
5.	Setting the Display Defaults	21
6.	Differences between VxGDB 1.0 and GDB 3.2	22

1. Introduction

The VxGDB 3.2 remote symbolic debugger is included in the Vx960 5.0.3 release. It provides source-level debugging of Vx960 applications from a UNIX™ workstation. VxGDB extends the line-oriented mode of the GNU Source-Level Debugger (GDB), Version 3.2, with a graphical user-interface based on the X Window System.

1.1 Overview

While running on the host system, VxGDB allows you to spawn and debug tasks running on networked Vx960 targets. Already-running tasks spawned from the Vx960 shell can also be debugged. To help diagnose and locate bugs, VxGDB provides extensive control and information features, allowing you to stop, resume, or alter task execution and to examine task and system resources such as the stack, data, symbol table, and source code. These features are highlighted below:

- **Stopping Task Execution**

VxGDB can stop task execution in four basic circumstances: (1) when encountering a breakpoint, (2) when single-stepping through the program, (3) when an exception occurs in the debugged task, or (4) when ^C is typed.

- (1) *Breakpoints* – You can set breakpoints at specific line numbers in source files, at entries to functions, or at specific addresses. You can also set temporary breakpoints, which are disabled when hit, and conditional breakpoints, which stop a task at a specified point only if a C language expression evaluates to true. Furthermore, you can specify commands to be automatically executed at a breakpoint. (For example, you might want to print the values of certain expressions.) You can disable, re-enable, and delete breakpoints at any time.
- (2) *Single-stepping* – You can step through a task, stopping after executing a single line of source or assembly code.

- (3) *Exceptions* – VxGDB will trap any exceptions (e.g., divide-by-zero) occurring within the debugged task.
- (4) *User Intervention* – You can suspend execution of the remote task by typing the interrupt character (usually ^C) in the dialogue window while the task is running.

- **Information Utilities**

When a task stops, a variety of VxGDB information utilities are available to help monitor the task and diagnose bugs. These utilities include commands to examine the stack, symbol table, source code, memory, variables, constants, and other data.

- (1) *Examining the stack* – When a task is stopped, you can backtrace through stack frames to see how the program arrived at its present state. Each frame of the stack contains data associated with a single call to a single function, including the function's local variables, and the address at which the function is executing.
- (2) *Examining the symbol table* – VxGDB provides information about the symbols (names of variables, functions, and types) defined in a program. It can list the names and data types of variables and functions, and describe where a variable is stored.
- (3) *Examining source files* – When the location of a bug has been determined, you can selectively list source lines by line number, function, or program address, and search forward or backwards for a text pattern in a source file.
- (4) *Examining data* – VxGDB can compute and print the value of any variable or C expression, including function calls, conditional expressions, casts, string constants, and values in memory and machine registers. To display the value of an expression frequently, you can add the expression to an automatic display list. Every time the task stops execution, the automatic display prints out expressions from the list and their values. You can examine previously displayed values with the VxGDB value history, which saves all printed values. You can also store values in "convenience variables," which hold a value for reference but have no effect on a task's execution.

- **Resuming or Altering Execution**

After examining the stack, source, data, or other information, you can continue execution of a stopped task, starting from where the task stopped. You can also alter execution of the task without editing source code, re-compiling, linking, or downloading. You can change the value of any local or global variable. You can force a function call to return immediately and you can set the return values. VxGDB also provides a jump command which allows you to continue the execution of a task at a line number or address other than the stopping point.

- **User-Interface**

The host portion of VxGDB consists of three components: a graphical user-interface, a command-line debugger, and a command driver.

- **xvxd**

The *graphical user-interface*, based on the X Window System (X11R4), lets you enter commands either by selecting a command button with the mouse or by typing the command directly into the dialogue window.

- **vxgdb960**

VxGDB960 is the *command-line debugger* based on the original GNU 3.2 version of `gdb`. It includes features such as command history and command-line editing. With this debugger, you can define new commands and command files made up of VxGDB command sequences that can be executed. VxGDB even provides a facility for documenting user-defined commands.

- **vxgdb**

The *command driver* invokes VxGDB with either the graphical user-interface or the line-oriented debugger. See 2.2 Invoking VxGDB for more information.

- **Vx960 Tools**

While using VxGDB, you can continue to take advantage of Vx960's native development tools. The combination of the Vx960 shell, symbolic debugging and disassembly, and performance monitoring facilities, provides a comprehensive high-level debugging solution. See 4.6 Using VxGDB with the Vx960 Shell Debugger for more information.

1.2 Documentation

The sections below describe the use of VxGDB with Vx960 and will familiarize you with the graphical user-interface to the debugger. A "getting started" section is included, as well as a discussion of how the display defaults can modify the appearance and behavior of the graphical user-interface.

Detailed information about the debugger command set can be found in Part 2, the Free Software Foundation's *GDB Manual, The GNU Source-Level Debugger, Third Edition, for GDB Version 3.2*. Refer to 6. Differences between VxGDB 1.0 and GDB 3.2 for any variations to the commands or functionality described in the *GDB Manual*.

Except for general page layout, the *GDB Manual* is supplied verbatim as distributed by the Free Software Foundation (FSF).

2. Getting Started with VxGDB

2.1 VxGDB Configuration

VxGDB 3.2 remote symbolic debugger is included in the Vx960 5.0.3 release. It provides source-level debugging of Vx960 applications from a UNIX workstation.

Please note the following important points:

- VxGDB 3.2 is configured in Vx960.
The `gdb` tools are found in the Vx960 bin directory; e.g., `/usr/vx/bin/sun4` or `/usr/vx/bin/ap400`. The remote server library `rdb.a` is already installed in the Vx960 libraries. `#define INCLUDE_RDB` is turned on in `/usr/vx/config/all/configAll.h`. See the *Vx960 Release Notes: Vx960 5.0.3* for additional information.
- The command-line version of `gdb` is supported on all hosts.

- There is a new X interface for the following hosts:
 - Sun-4
 - Sun-3
 - HP9000/300
 - IBM RS/6000
- Source code for `gdb` is not supplied on this tape. If you want source code, request it from your local Intel sales office, or send a message to `vx960bugs@ichips.intel.com`. Intel will provide any source code derived from FSF.

There are two `gdb` front-ends. There is a command-line version called `vxgdb960` that is based on the original GNU 3.2 version. There is also a new X11R4 based version, called `xvxgdb` which provides visual debugging via an X client. The interface is similar to Sun's `dbxtool`.

For example, assume that `/usr/vx` is the name of your Vx960 directory, and that you will be running VxGDB on a Sun 4 host system. If your shell is `csh`, you would use the command:

```
% setenv PATH /usr/vx/bin/sun4:$PATH
```

or the following:

```
% set path=(/usr/vx/bin/sun4 $path)
```

In `sh` or `ksh`, use the command:

```
$ PATH=/usr/vx/bin/sun4:$PATH; export PATH
```

Put these commands in the appropriate shell startup file (e.g., `.cshrc` or `.profile`).

VxGDB consists of code that runs on both the UNIX host and the Vx960 target. The remote debug server (RDB) is installed on the Vx960 target. This server resides in the Vx960 library `rdb.a` and is incorporated into the system image when source-level debugging is enabled in the Vx960 configuration by defining `INCLUDE_RDB` in the configuration file `configAll.h`. Defining `INCLUDE_RDB` adds the RDB interface routines and spawns the source debugging task `tRdbTask` when Vx960 is booted. For more information on configuring and remaking Vx960, see 8. Configuration, in the *Vx960 Programmer's Guide*.

2.2 Invoking VxGDB

VxGDB is invoked on the host command line with the following syntax:

```
% vxgdb [-1] [ gdb-option... ] [ gui-option... ]
```

The following are two examples of the above syntax:

```
vxgdb
```

or

```
vxgdb -1
```

NOTE: Both `vxgdb960` and `vxgdb -1` invoke the command-line VxGDB.

If the `-1` (command-line) flag is used, it must appear as the first option. You can also set the shell variable `VXGDB_DEBUGGER` to specify the name of the VxGDB executable file; its value supersedes any other specification of the debugger executable file.

VxGDB also accepts the GDB command-line options (*gdb-option*), and the standard Xtoolkit command-line options (*gui-option*)¹.

When VxGDB starts up, it looks for the file `.vxgdbinit` in your home directory. If it finds the file, VxGDB executes the commands (one per line). VxGDB searches next for the file `.vxgdbinit` in the current working directory, and (if found) executes the commands in this file. Refer to the section entitled **13. Canned Sequences of Commands** in the *GDB Manual* for more information on command files.

3. VxGDB Graphical User-interface

VxGDB uses a graphical user-interface based on the X Window System. You can invoke a VxGDB command by using a mouse to select a command button in the command window or by typing the equivalent command in a dialogue window. The VxGDB windows and command buttons are described below.

1. For more information, refer to *X Toolkit Intrinsic - C Language Interface, X Window System, X Version 11, Release 4*, Massachusetts Institute of Technology, 1988.

3.1 Windows

The VxGDB graphical user-interface consists of the following windows:

- | | |
|-----------------|---|
| File Window | - Displays the full pathname of the file currently displayed in the source window, and the line number in the file where the insertion point (displayed as a caret in the source window) is located. The file window is noted as ❶ in Figure 1 |
| Source Window | - Displays the current source file. You can move through the source file with the scrollbar, which appears on the left side. You can also move through a file with the cursor keys (if the keyboard and X11R4 display server support cursor movements). The source window is noted as ❷ in Figure 1. |
| Message Window | - Displays the last message issued by the debugger. The message window is noted as ❸ in Figure 1. |
| Command Window | - Contains a set of buttons for frequently used VxGDB commands. Clicking the left mouse button while the pointer is inside a button activates the associated VxGDB command. Some buttons use information selected in the source window as an implicit argument to the debugger command. Some buttons exhibit different behavior when they are selected by the right mouse button (see 3.3 Command Buttons). The command window is noted as ❹ in Figure 1. |
| Dialogue Window | - When the pointer is in this window, any keystroke you type (before typing RETURN) serves as direct input to the debugger. Note that the GDB command-line editing and history expansion features are disabled when VxGDB runs under the graphical user-interface (see 2. GDB User-interface in the <i>GDB Manual</i>). The dialogue window is noted as ❺ in Figure 1. |
| Display Window | - When you enter the display command, the values of variables are collected and displayed in this window each time execution stops. Use the undisplay command to stop reporting the value of variables. The display window is noted as ❻ in Figure 1. |

WINDOW TYPE

The screenshot displays the VxGDB 1.0.1 Beta-3 interface with the following components:

- File:** Shows the current file as `usr/test/prog.c` at line 105.
- Source:** Contains the following C code:


```

      )
      /*-----*/
      * setBoo - low level routine
      */
      VOID setBoo (pBoo)
      struct boo *pBoo:
      {
      pBoo->theInt = 99;
      pBoo->c = '1';
      pBoo->next = (struct boo *) malloc (sizeof (struct boo));
      bzero ((char *) pBoo->next, sizeof (struct boo));
      pBoo->next->theInt = 999;
      pBoo->next->c = '+';
      pBoo->next->next = 0;
      }
      
```
- Message:** Displays the instruction: "Click the label to pop down the data popup".
- Command:** A row of buttons for debugging actions: `cont`, `next`, `step`, `finish`, `break`, `thbreak`, `delete`, `up`, `down`, `print`, `print #`, `display`, `undisplay`, `args`, `locals`, `stack`, `search`, and `quit`.
- Dialogue:** Shows the execution flow:


```

      (vxgdb)
      Continuing.
      Bpt 3. setBoo (pBoo=(struct boo *) 0x1aba30) (prog.c line 102)
      (vxgdb) (vxgdb) next
      (vxgdb) next
      (vxgdb) next
      (vxgdb) (vxgdb)
      (vxgdb) _
      
```
- Display:** Shows the current state of the `pBoo` pointer:


```

      1: *pBoo = {
      theInt = 99,
      c = 124 '1',
      next = 0x1be968
      }
      
```

Figure 1. VxGDB Windows

3.2 Window Operations

You can adjust the relative sizes of the source window, command window, dialogue window, and display window by dragging the grip (a small rectangle near the right edge of a horizontal border) with the left mouse button held down.

When there is a scrollbar present, pressing the left mouse button scrolls the text forward in a window while pressing the right mouse button scrolls the text backward. The amount of scrolling depends on the distance of the pointer button away from the top of the scrollbar. If the button is pressed at the top of the scrollbar, only one line of text is scrolled. If the button is pressed at the bottom of the scrollbar, one screenful of text is scrolled.

Pressing the middle mouse button down on the scrollbar and dragging will dynamically cause the text to slide around under the window for quick positioning.

3.3 Command Buttons

You can invoke many frequently used VxGDB commands using command buttons in the command window. Some command buttons use the selection made in the source window as an argument when presenting the command to the underlying debugger for execution.

For example, you can move the insertion point (caret) in the source window to a particular line and then *click* (press and release) on the `(break)` button. The source line containing the insertion point serves as the argument to the `(break)` button for VxGDB, and a breakpoint is set at the marked place. You can also move the pointer to a function name, click the left mouse button once, and the function name is highlighted. Clicking the `(break)` button causes a breakpoint to be set at the first executable line of the function. In the following discussion, *selection* refers to either highlighted text in the source window or the position of the insertion point.

Text selection in the source window has been modified to make it easier to select C expressions. Clicking the left mouse button selects a C expression by highlighting it in reverse-video. This also positions the insertion point (caret) and updates the file window accordingly.

C expression selection is based on the X11R4 resource *delimiters* which determine the set of characters that delimit a C expression. Text selection adjustment is possible by holding down the left mouse button and dragging the caret to extend the highlighted text.


Clicking the left mouse button while holding down the SHIFT key prints the value of the expression selected.

The debugger commands that can be invoked with a command button are listed below by function. Those commands that use the current selection are marked with the symbol ✓.

3.3.1 Execution Command Buttons

- Continue execution from where the target program stopped.
- Execute one source line, stepping into a function if the source line contains a function call.
- Execute one source line, stepping over function calls.
- Continue execution until the currently selected stack frame returns.²

3.3.2 Breakpoint Command Buttons

- ✓ - Set a breakpoint at the line where program execution is to be halted. Place the caret on the source line and click on the button. A breakpoint symbol will appear next to the source line. You can also click on a function name using the left mouse button and then click on the button; the program will then be stopped at the first executable line of the selected function. A breakpoint symbol will be placed at that location. An arrow symbol next to the breakpoint symbol indicates that execution has stopped at the indicated line. The breakpoint and arrow symbols are illustrated in the source window  in Figure 1.
- ✓ - Like the command button, except that a temporary breakpoint is placed at the indicated location.

2. *currently selected* in this context does not refer to the current text selection, but to the notion in GDB of the selected stack frame (refer to the section entitled 8. Examining the Stack of the *GDB Manual* (Part 2)).

- `delete` ✓ - Remove the breakpoint named by the selected breakpoint number, or set at the selected source line, function, or breakpoint number.

3.3.3 Stack Command Buttons

- `up` - Move up one level on the call stack.
- `down` - Move down one level on the call stack.
- `stack` - Show a stack trace of the functions called.
- `locals` - Show the local variables of the currently selected frame.²
- `args` - Show the arguments of the currently selected frame.²

NOTE: For information on the limitations to the data you will receive using the `locals` and `args` buttons, refer to **10.4.1 Task Trace:tt()** of the *Vx960 Programmer's Guide*.

3.3.4 Data Display Command Buttons

- `print` ✓ - Print the value of the selected expression.
- `print *` ✓ - Print the value of the object the selected expression is pointing to.
- `display` ✓ - Display the value of a selected expression in the display window, updating its value every time execution stops.
- `undisplay` ✓ - Delete the display associated with the selected number. (You can select the number of the display to remove directly from the display window).

3.3.5 Miscellaneous Command Buttons

- `search` - Pop up a search panel which allows both forward (>>) and reverse (<<) search of text strings in the source file. Typing

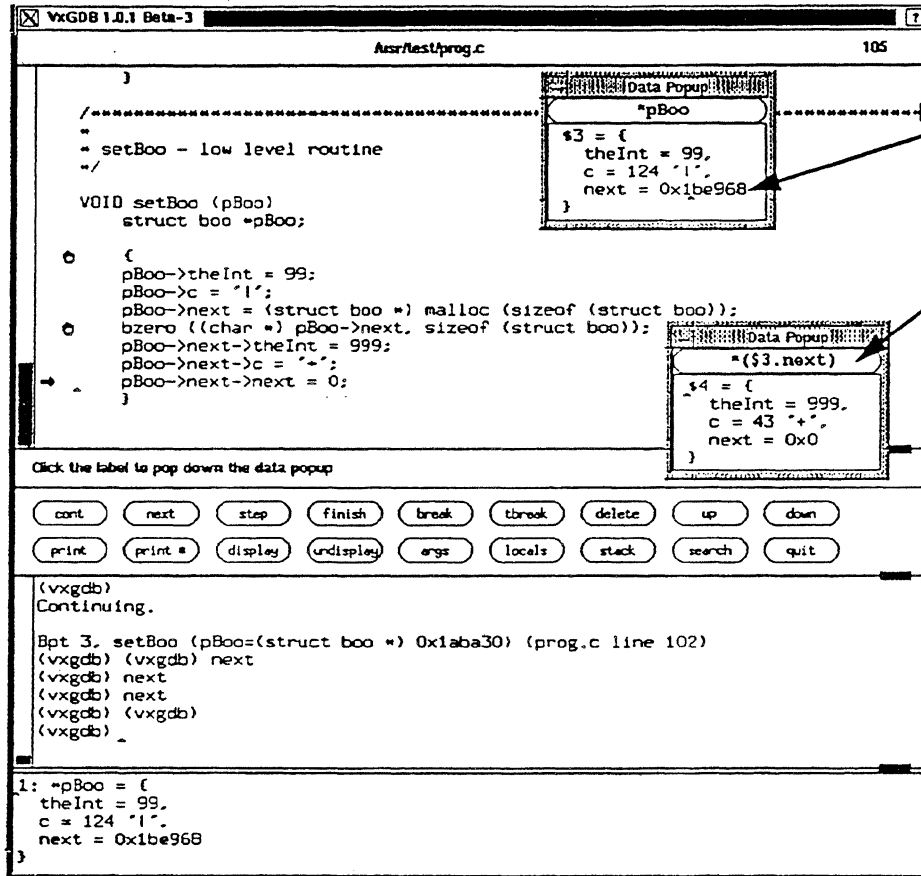
RETURN after entering the search string will begin a forward search and pop down the search panel.

`quit`

- Exit VxGDB.

3.4 Displaying C Data Structures

VxGDB provides support for graphically displaying C structures and following pointers to linked data structures. Clicking the right mouse button on the `print` button (or the `print *` button) displays the value of the selected expression (or the value of the selected expression being referenced) in a *data popup* window. If the value is a pointer or a structure containing pointers, you can examine the value of the referenced object by clicking the left mouse button on the pointer value (see ❶ in Figure 2). This will create another data popup window that displays the object the pointer points to. Pressing the label of a data popup window will remove it and all of its descendants (see ❷ in Figure 2).



- 1 Click on the value and hold to traverse to the next element in the list.
- 2 Click here to remove the popup window.

Figure 2. Displaying C Structures

4. Debugging with VxGDB

4.1 Specifying Source Directories

The symbol table for an object module records the name of the source file from which it was compiled, but not the directory in which the source file resides. VxGDB maintains a list of directories to search for source files; this list is called the "source path." Whenever VxGDB needs to display source code, it searches all the directories in the source path in order, until it finds a source file with the desired name.

When VxGDB starts up, the source path consists of the current working directory only. To add other directories, use the `directory` command.

Another way to add directories to the source path is to use the `-d` flag when invoking VxGDB. See the section 4. Options and Arguments for GDB in the *GDB Manual*.

To see VxGDB's current source path, type in the dialogue window:

```
(vxgdb) info dir
```

4.2 Connecting to a Vx960 Target

The VxGDB command `target` lets you connect to a Vx960 target on the network. To connect to a target whose host name is "tt", type in the dialogue window:

```
(vxgdb) target tt
```

VxGDB will display a message such as:

```
Attaching remote machine across net...  
Connected to tt.
```

VxGDB will then attempt to read the symbol tables of any object modules loaded into the Vx960 target since it was last booted. VxGDB will locate the object files by searching the directories listed in the source path; if it fails to find an object file, it will display a message such as:

```
prog.o: No such file or directory.
```

This will not cause the target command to abort. VxGDB will print an error message in this case, but will continue with the execution of the target command. However, you will not be able to debug a program that references the object modules VxGDB was unable to find. You may use the `directory` command to specify the location of these object modules and reissue the target command.

4.3 Compiling an Application Module

Compile source files as you do normally for Vx960, but add the `-g` option, which produces additional symbol table information for debuggers. For example, to compile the demo program `prog.c` provided in `/usr/vx/demo/rdb`³, type from the UNIX host:

```
% cd /usr/vx/demo/rdb
% gcc960 -DCPU=I960CA -ACA -c -g -I/usr/vx/h prog.c
```

4.4 Downloading to a Vx960 Target

If you have connected to the Vx960 target and you want to debug an object file that has not yet been loaded, you can use the VxGDB `load` command to download a file from UNIX to Vx960 incrementally. The object file given as an argument to the `load` command is actually opened twice: first by the Vx960 target in order to download the code, then by VxGDB in order to read the symbol table. This can lead to problems if the current working directories on the two systems differ. It is simplest to set the working directory on both systems to the directory in which the object file resides, and then to reference the file by its name, without any path. Thus, to load `prog.o`, the example demo program, type from the Vx960 prompt:

```
-> cd "/usr/vx/demo/rdb"
```

From VxGDB, type in the dialogue window:

```
(vxgdb) cd /usr/vx/demo/rdb
(vxgdb) directory /usr/vx/demo/rdb
(vxgdb) load prog.o
```

VxGDB will display a response like:

```
Reading symbol data from /usr/vx/demo/rdb/prog.o... done.
```

3. For clarity, in examples where a full pathname must be specified, the files are often referred to as `/usr/vx/filename`. However, Vx960 does not assume or require this pathname.

You can also use the `load` command to reload an object module after editing and re-compiling the corresponding source file. Note that this will cause VxGDB to delete all currently defined breakpoints, auto-displays, and convenience variables, and to clear the value history. (This is necessary in order to preserve the integrity of debugger data structures that reference the target system's symbol table.)

Avoid using the `load` command to reload an object module while debugging a task. The debugger will not have accurate symbol information if the task being debugged makes references to the reloaded object module, and all breakpoints and auto-displays set for the task will be deleted. Use the `kill` command to delete the current task before reloading an object module.

4.5 Starting to Debug

After you have configured VxGDB (refer to 2.1 VxGDB Configuration), you are ready to run the following tutorial. A supplied sample program, `prog.c`, is used as an example throughout this section.

- (1) Copy the sample file `/usr/vx/demo/rdb/prog.c` to a local directory like `/my/directory`. Now compile the file using the appropriate compiler with the `-g` command-line option to produce the file `prog.o`:

```
% cd /my/directory
% cp /usr/vx/demo/rdb/prog.c .
% gcc960 -DCPU=I960CA -ACA -c -g -I/usr/vx/h prog.c
```

- (2) Log in to your target machine and load the new object module into the target's memory.

```
-> cd "/my/directory"
-> ld < prog.o
```

- (3) From your UNIX host, change to the directory that contains `prog.o` and type:

```
% vxgdb
```

- (4) From the command-line:

```
(vxgdb) target tt
```

If all is well, you should see a message indicating that symbols have been loaded from `prog.o`.

- (5) To activate the source window, type in the dialogue window:

```
(vxgdb) break prog
```

to set a breakpoint. The source code in `prog.c` should immediately appear in the source window, and you should see a breakpoint symbol next to the first executable line of the main procedure.

- (6) To run the program, type in the dialogue window:

```
(vxgdb) run prog 10
```

Note that when spawning a task under Vx960, you must specify an entry point to run; unlike GDB, `main` is not assumed. You should now see an arrow symbol \blacktriangleright next to the breakpoint symbol in the source window; this means that execution has stopped at the indicated line.

- (7) Using the mouse, click on the `step` button a few times. You will see the arrow symbol advance to follow the execution of the program.
- (8) Click on a variable within scope and click on the `print` or `print *` button. If you use the `display` command to display a variable, its value will appear in the bottom window. You can resize this window if the full value of the variable cannot be seen. The scrollbar is also available if you are displaying more variables than can comfortably fit on the screen.
- (9) Click on a pointer to a structure, and use the right button to click on the `print *` button. You should see a structure-traversal window appear. Clicking on any pointer's value should advance to the indicated member of the structure.

For an extensive explanation of the debugger command set, see the *GDB Manual*, Part 2 of this publication.

4.6 Using VxGDB with the Vx960 Shell Debugger

While using VxGDB, you can continue to take advantage of the Vx960 shell for symbolic debugging and disassembly, and performance monitoring facilities. The combination provides a comprehensive high-level debugging solution. The interaction between VxGDB and the Vx960 shell debugger is described below.

4.6.1 Setting Breakpoints in the Same Task

You can set breakpoints from either VxGDB or the Vx960 shell, but they should not access the same task. If a breakpoint set from the Vx960 shell is encountered by VxGDB, task execution will stop and the following error message will be displayed:

```
Unexpected breakpoint/trace event
```

To avoid this problem, make sure that breakpoints set from VxGDB and the Vx960 shell operate on different tasks. Note that any breakpoint set on all tasks from the Vx960 shell can affect a task being debugged by VxGDB.

4.6.2 Monitoring Spawned Tasks

You can monitor a task spawned by VxGDB from the Vx960 shell, or monitor a task spawned by the Vx960 shell from VxGDB.

When you start a program from VxGDB using the run command, a task named *tRdbRun* is spawned on the Vx960 target system. You can change its priority, task options, and stack size from the Vx960 shell by changing the value of the task variables *rdbRunTaskPriority*, *rdbRunTaskOptions*, and *rdbRunTaskStackSize*. The default values are:

```
rdbRunTaskPriority    100  
rdbRunTaskOptions    VX_SUPERVISOR_MODE | VX_FP_TASK | VX_STDIO  
rdbRunTaskStackSize 20000
```

To use VxGDB to debug a task spawned from the Vx960 shell, invoke the attach command by typing in the dialogue window:

```
(vxgdb) attach taskId
```

where *taskId* is the Vx960 hexadecimal task ID. The task can be running or suspended when you attach. If running, the task will be suspended at the time of attachment.

5. Setting the Display Defaults

Many attributes of the display interface can be changed with the X11R4 defaults mechanism. The following examples illustrate changes to VxGDB that can be made from your `.Xdefaults` file:

- Change the debugger invoked by default:

```
xvxgdb*debugger: /usr/tools/intel/VxGDB.960
```

- Change the font in the source window to bold, the font in the dialogue window to Courier, and the font in the message window to Times-Roman:

```
xvxgdb*sourceWindow*font: 9x15bold
```

```
xvxgdb*dialogWindow*font: -adobe-courier-medium-r-normal--18-180-75-75-m-110--*-1
```

```
xvxgdb*messageWindow*font: -*--times-medium-r-*--*-180-*--*-*-*-*
```

- Change the shape and highlight thickness of the command buttons:

```
xvxgdb*Command.shapeStyle: Rectangle
```

```
xvxgdb*Command.highlightThickness: 4
```

- Change the size and position of the window pane grips for the main window. If the display is a color monitor, change the color of the grips to green:

```
xvxgdb*Paned*gripIndent: 5
```

```
xvxgdb*Grip.height: 5
```

```
xvxgdb*Grip.width: 5
```

```
xvxgdb*Grip.foreground: green
```

- Change the key bindings in the source window so that `^F` means “forward page,” as in the `vi` editor:

```
xvxgdb*sourceWindow.Translations: #override\  
Ctrl<Key>F: next-page ()
```


6. Differences between VxGDB 1.0 and GDB 3.2

Detailed information about the debugger command set can be found in Part 2, the Free Software Foundation's *GDB Manual, The GNU Source-Level Debugger, Third Edition, for GDB Version 3.2*. Some of the commands documented in Part 2 have been disabled in VxGDB since they have no meaning when debugging a remote Vx960 target system. The disabled commands are:

add-file	core-file
exec-file	handle
info environment	info signals
set environment	signal
tty	unset environment

NOTE: The add-file command is replaced by the load command in VxGDB.

VxGDB does not handle signals, but will trap any exceptions (e.g., divide-by-zero) within the debugged task.

In addition, the following commands described in the *GDB Manual* behave differently in VxGDB:

run - The arguments to the VxGDB run command are interpreted by the remote debug server running on the Vx960 target, rather than the UNIX shell as is the case in GDB. This means that UNIX shell meta-characters or environment variables appearing in the argument list are not expanded. Items that can appear in the argument list are as follows:

- C-style hexadecimal and decimal constants
- C-style string constants
- C-style character constants
- Input/output direction operators < and >
- Names of symbols in the Vx960 system symbol table

- symbol-file**
- The **symbol-file** command is used internally by VxGDB to load the symbol table of the Vx960 system image from which the target was booted. This command should not be issued directly. Use the load command to load object modules incrementally.

