**What is GDB..?**

**What is DEBUGGING..?**

**Types of debugging..?**

**Remote debugging / tcp debugging.**

**Serial debugging.**

**Building kernel image adding gdb tool chain via busy box..**

**Building a kernel image for target board.**

**-via build root.**

**-via crosstool compilation.**

GDB
STOP BUGS DEAD

GNU

What is GDB?

GDB stands for GNU Debugger. It is a powerful command-line tool for debugging programs written in various programming languages, including C, C++, Ada, Fortran, and others. GDB allows developers to examine and modify the execution of their programs, set breakpoints, and step through code to track down errors and bugs.

Some of the key features of GDB include the ability to:

- Set breakpoints at specific lines of code or functions
- Inspect the values of variables and memory locations
- Step through code one instruction at a time
- View and manipulate the call stack
- Attach to running processes for debugging
- Debug programs on remote machines over a network

GDB is widely used in the development of complex software projects and is considered an essential tool for debugging programs at a low level. However, it requires some familiarity with the command-line interface and a basic understanding of the architecture of the program being debugged.

To use GDB, you typically start by compiling your program with debugging symbols, which allow GDB to map the executable code back to the original source code. You then run GDB and provide it with the name of the executable file to debug. From there, you can use a variety of commands to set breakpoints, inspect variables, and step through the code.

Overall, GDB is a powerful tool that can help developers track down and fix bugs in their code, making it an essential part of any software development toolkit.

GDB offers a big list of commands, however the following commands are the ones used most frequently:

- **b main** - Puts a breakpoint at the beginning of the program
- **b** - Puts a breakpoint at the current line
- **b N** - Puts a breakpoint at line N
- **b +N** - Puts a breakpoint N lines down from the current line
- **b fn** - Puts a breakpoint at the beginning of function "fn"
- **d N** - Deletes breakpoint number N
- **info break** - list breakpoints
- **r** - Runs the program until a breakpoint or error
- **c** - Continues running the program until the next breakpoint or error
- **f** - Runs until the current function is finished
- **s** - Runs the next line of the program
- **s N** - Runs the next N lines of the program
- **n** - Like s, but it does not step into functions
- **u N** - Runs until you get N lines in front of the current line
- **p var** - Prints the current value of the variable "var"
- **bt** - Prints a stack trace
- **u** - Goes up a level in the stack
- **d** - Goes down a level in the stack
- **q** - Quits gdb

## Getting Started: Starting and Stopping

- gcc -g myprogram.c
  - Compiles myprogram.c with the debugging option (-g). You still get an a.out, but it contains debugging information that lets you use variables and function names inside GDB, rather than raw memory locations (not fun).

- gdb a.out
    - Opens GDB with file a.out, but does not run the program. You'll see a prompt (gdb) - all examples are from this prompt.
- r
- r arg1 arg2
- r < file1
    - Three ways to run "a.out", loaded previously. You can run it directly (r), pass arguments (r arg1 arg2), or feed in a file. You will usually set breakpoints before running.
- help
- h breakpoints
    - Lists help topics (help) or gets help on a specific topic (h breakpoints). GDB is well-documented.
- q - Quit GDB

## Stepping through Code

Stepping lets you trace the path of your program, and zero in on the code that is crashing or returning invalid input.

- l
- l 50
- l myfunction
    - Lists 10 lines of source code for current line (l), a specific line (l 50), or for a function (l myfunction).
- next
    - Runs the program until next line, then pauses. If the current line is a function, it executes the entire function, then pauses. **next** is good for walking through your code quickly.
- step
    - Runs the next instruction, not line. If the current instruction is setting a variable, it is the same as **next**. If it's a function, it will jump into the function, execute the first statement, then pause. **step** is good for diving into the details of your code.
- finish
    - Finishes executing the current function, then pause (also called step out). Useful if you accidentally stepped into a function.

## Breakpoints or Watchpoints

Breakpoints play an important role in debugging. They pause (break) a program when it reaches a certain point. You can examine and change variables and resume execution. This is helpful when some input failure occurs, or inputs are to be tested.

- break 45
- break myfunction
  - Sets a breakpoint at line 45, or at myfunction. The program will pause when it reaches the breakpoint.
- watch x == 3
  - Sets a watchpoint, which pauses the program when a condition changes (when x == 3 changes). Watchpoints are great for certain inputs (myPtr != NULL) without having to break on *every* function call.
- continue
  - Resumes execution after being paused by a breakpoint/watchpoint. The program will continue until it hits the next breakpoint/watchpoint.
- delete N
  - Deletes breakpoint N (breakpoints are numbered when created).

## Setting Variables

Viewing and changing variables at runtime is a critical part of debugging. Try providing invalid inputs to functions or running other test cases to find the root cause of problems. Typically, you will view/set variables when the program is paused.

- print x
  - Prints current value of variable x. Being able to use the original variable names is why the (-g) flag is needed; programs compiled regularly have this information removed.
- set x = 3
- set x = y
  - Sets x to a set value (3) or to another variable (y)
- call myfunction()
- call myotherfunction(x)
- call strlen(mystring)
  - Calls user-defined or system functions. This is extremely useful, but beware of calling buggy functions.
- display x
  - Constantly displays the value of variable x, which is shown after every step or pause. Useful if you are constantly checking for a certain value.
- undisplay x
  - Removes the constant display of a variable displayed by display command.

## Backtrace and Changing Frames

A stack is a list of the current function calls - it shows you where you are in the program. A *frame* stores the details of a single function call, such as the arguments.

- bt
    - **Backtraces** or prints the current function stack to show where you are in the current program. If main calls function a(), which calls b(), which calls c(), the backtrace is
- c <= current location
  b
  a
  main

- up
- down
    - Move to the next frame up or down in the function stack. If you are in **c,** you can move to **b** or **a** to examine local variables.
- return
    - Returns from current function.

## Handling Signals

Signals are messages thrown after certain events, such as a timer or error. GDB may pause when it encounters a signal; you may wish to ignore them instead.

- handle [signalname] [action]
- handle SIGUSR1 nostop
- handle SIGUSR1 noprint
- handle SIGUSR1 ignore
    - Instruct GDB to ignore a certain signal (SIGUSR1) when it occurs. There are varying levels of ignoring.

Go through the following examples to understand the procedure of debugging a program and core dumped.

- Debugging Example 1
- This example demonstrates how you would capture an error that is happening due to an exception raised while dividing by zero.
- Debugging Example 2
- This example demonstrates a program that can dump a core due to non-initialized memory.

Both the programs are written in C++ and generate core dump due to different reasons. After going through these two examples, you should be in a position to debug your C or C++ programs generating core dumps.

What is Debugging?

Debugging is the process of locating and removing coding mistakes in computer programs. In information technology and engineering, the word 'bug' is a synonym for the word 'error.' The goal of debugging is to identify and correct an error's root cause.

Debugging plays an important role in the software development process and ironically, testing to determine and eliminating the presence of bugs can take just as much time as writing code. The debugging process itself consists of identify the cause of an error and fixing it. During the debugging process, which can be carried out manually or automated through software debugging tools, engineers will look for:

> RunTime errors
> Errors in logic
> Implementation errors
> False outputs

For Linux and Unix operating systems, GDB is used as a standard debugger.

For Windows OS, the visual studio is a powerful editor and debugger.

For Mac OS, LLDB is a high-level debugger.

Debugging is an essential skill for software developers, system administrators, and anyone who works with complex computer systems. It often requires a combination of analytical thinking, technical expertise, and creativity to identify and solve problems.

There are many techniques and tools used for debugging, such as logging, unit testing, code profiling, and interactive debugging tools. Debugging can be a time-consuming and challenging process, but it is a critical step in ensuring the reliability and quality of software and systems.

**Advantages of debugging:**

1. Helps in identifying and fixing errors: Debugging helps in identifying and resolving errors, bugs, or defects in software code or hardware systems. This helps in ensuring that the program or system works as intended and produces the expected results.
2. Improves software quality: Debugging helps in improving software quality by identifying and fixing issues before the software is released to users. This can help prevent crashes, data loss, and other issues that can negatively impact user experience.
3. Saves time and resources: Debugging helps in identifying and fixing issues early in the software development process, which can save time and resources in the long run. It can help prevent costly and time-consuming fixes later in the development process or after the software has been released.

**Disadvantages of debugging:**

4. Time-consuming: Debugging can be a time-consuming process, especially for complex systems. It requires a significant amount of time and effort to identify and resolve issues, which can delay software development and release timelines.
5. Can be difficult: Debugging can be a challenging process that requires specialized technical skills and knowledge. It can be difficult to identify the root cause of an issue, especially in complex systems.
6. Can be costly: Debugging can be costly, especially if the issues are discovered late in the development process or after the software has been released. Fixing issues after release can require significant resources, including time, money, and manpower.

**Difference between GDB and GDBSERVER?**

GDB (GNU Debugger) is a powerful debugging tool used for debugging programs written in various programming languages, including C, C++, and Assembly. GDB runs on a host machine and allows developers to debug programs running on a target machine.

GDB server, on the other hand, is a program that runs on the target machine and acts as an intermediary between the debugger (GDB) running on the host machine and the program being debugged on the target machine. GDB server provides a debugging interface on the target machine and handles communication between the host and target machines.

The main difference between GDB and GDB server is that GDB is a debugging tool that runs on the host machine, while GDB server is a program that runs on the target machine and provides a debugging interface for GDB to communicate with the program being debugged.

GDBServer, on the other hand, is a program that enables remote debugging with GDB. It allows a developer to debug a program on a remote system from a local machine using GDB. GDBServer listens on a specific port for incoming connections from GDB and provides a debugging interface to the target program running on the remote system.

Here are some key differences between GDB and GDBServer:

7. Purpose: GDB is a standalone tool used for debugging local programs, while GDBServer is used for remote debugging.
8. Platform: GDB can be used on both local and remote systems, while GDBServer is typically used on remote systems.
9. Interface: GDB provides a command-line interface, while GDBServer provides a network interface that allows remote debugging.
10. Location of target program: In local debugging with GDB, the target program is executed on the same system where GDB is running. In remote debugging with GDBServer, the target program is executed on a remote system.

In summary, GDB is used for debugging local programs, while GDBServer is used for remote debugging. While GDB provides a command-line interface for local debugging, GDBServer provides a network interface for remote debugging with GDB.

Types of Debugging..?

Print debugging: This involves inserting print statements in the code to display variable values or other information at specific points in the program's execution. It is a simple and effective debugging technique, but it can also be time-consuming and may require modifying the code.

Interactive debugging: Interactive debugging involves using a debugger tool like GDB (GNU Debugger) to interactively debug a program. Developers can set breakpoints, step through the code, inspect variables, and execute commands while the program is running. This technique is more efficient and flexible than print debugging and allows developers to debug complex issues.

Core dump analysis: When a program crashes, it generates a core dump file, which contains information about the program's state at the time of the crash. Developers can use tools like GDB to analyze the core dump file and identify the cause of the crash.

System call tracing: This technique involves tracing system calls made by a program to understand its behavior. Tools like strace and ltrace can be used to trace system calls and library calls made by a program and identify issues like resource leaks or incorrect system call usage.

5.Profiling: Profiling involves collecting data on a program's execution, including how much time is spent in each function or code block. Tools like gprof can be used to generate a profile report and identify performance bottlenecks or areas for optimization.

11. Unit testing: Unit testing involves the creation of test cases for individual code modules or functions. These tests are designed to ensure that each module or function works correctly and produces the expected results. Any errors or bugs discovered during unit testing can be addressed before they impact the broader system.
12. Integration testing: Integration testing involves testing the interaction between different modules or components of a software system. This helps identify errors or bugs that arise from the integration of different components.
13. Regression testing: Regression testing involves retesting software after making changes or updates to ensure that the changes have not introduced new errors or bugs.
14. Static analysis: Static analysis involves analyzing the code without executing it to identify potential issues, such as syntax errors, type errors, and other common programming mistakes.
15. Dynamic analysis: Dynamic analysis involves analyzing the behavior of a program as it executes to identify issues such as memory leaks, race conditions, and other issues that can only be detected at runtime.

Each type of debugging has its own strengths and weaknesses, and developers may use one or more of these techniques depending on the nature of the problem and the type of software being developed.

Importance of Debugging

Debugging is a critical part of the software development process, and its importance cannot be overstated. Here are some of the reasons why debugging is so important:

Finding and fixing errors: Debugging is the process of identifying and fixing errors or defects in a program. Errors can cause a program to crash, behave unexpectedly, or produce incorrect results. By debugging the code, developers can find and fix these errors, ensuring that the program works as intended.

Improving software quality: Debugging can help to improve the quality of software by identifying and fixing errors before the program is released to users. This can reduce the number of bugs that users encounter, making the program more reliable and easier to use.

Saving time and money: Debugging can save time and money by identifying and fixing errors early in the development process. The cost of fixing errors increases as the project progresses, so identifying and fixing errors early can save time and money in the long run.

Facilitates software testing: Debugging is an essential part of software testing, as it helps to identify issues that need to be addressed before software is released. Debugging can also help to identify areas that need further testing to ensure software quality.

Enhancing performance: Debugging can help to identify performance bottlenecks in a program, allowing developers to optimize the code and improve its performance.

Increasing developer productivity: Debugging can increase developer productivity by helping them to understand how the code works and how to improve it. By debugging the code, developers can learn more about the program's behavior and improve their programming skills.

Debugging Strategies:

Debugging strategies are methods and techniques that developers use to identify and resolve issues in software code or hardware systems. These strategies are designed to help developers find and fix bugs, defects, and errors in the software code or hardware, which can help improve the overall quality and performance of the system.

There are various debugging strategies available, including:

16. Divide and conquer: This strategy involves breaking down the problem into smaller, more manageable pieces and isolating the issue by testing each piece individually. By narrowing down the problem, developers can identify the specific area of the code that is causing the issue.
17. Incremental testing: Incremental testing involves testing software as it is being developed, rather than waiting until the entire system is complete. This allows developers to identify issues early in the development process and address them before they become more significant.
18. Rubber duck debugging: This technique involves explaining the code to an inanimate object, such as a rubber duck, to help identify issues. By explaining the code in detail, developers may spot errors or issues that they may have missed otherwise.
19. Print statements: Adding print statements to the code can help developers identify the specific area of the code that is causing the issue. This can help narrow down the problem and identify the root cause of the issue.
20. Debugging tools: Debugging tools such as debuggers, memory analyzers, and profiling tools can help developers identify and resolve issues quickly and effectively.
21. Pair programming: Pair programming involves two developers working together on the same code. This can help identify issues more quickly, as both developers can work together to identify and resolve issues.
22. Code review: Code review involves having other developers review the code for issues or potential problems. This can help identify issues that the original developer may have missed.

Each debugging strategy has its own strengths and weaknesses, and developers may use one or more of these strategies depending on the nature of the problem and the type of software being developed.

Debugging Tool

There are many tools available to developers for debugging their code. Here are some of the most popular debugging tools:

GDB: GDB (GNU Debugger) is a powerful command-line debugger that can be used to debug programs written in C, C++, and other languages. It allows developers to set breakpoints, step through code, examine variables, and more.

Valgrind: Valgrind is a suite of debugging and profiling tools that can help developers find memory leaks, race conditions, and other errors in their code. It includes tools like Memcheck, which can detect memory errors, and Helgrind, which can detect threading errors.

strace: strace is a command-line tool that can be used to trace system calls and signals made by a program. It can help developers identify system-related issues and performance bottlenecks.

Wireshark: Wireshark is a network protocol analyzer that can be used to debug network-related issues. It allows developers to capture and analyze network traffic, which can help them identify and fix issues related to network connectivity and performance.

5. Visual Studio Debugger: Visual Studio Debugger is a debugging tool that is included with Microsoft's Visual Studio IDE. It provides a graphical user interface that allows developers to set breakpoints, step through code, examine variables, and more.

6. Eclipse Debugger: Eclipse Debugger is a debugging tool that is integrated into the Eclipse IDE. It provides similar functionality to the Visual Studio Debugger, including the ability to set breakpoints, step through code, and examine variables.

A debugging tool is a software application or program that helps developers identify and fix issues in their code. These tools are designed to provide developers with insights into the behavior of their code, including how it executes, what resources it uses, and how it interacts with other parts of the system.

Some popular debugging tools include:

23. Integrated development environments (IDEs): IDEs are software applications that provide developers with a comprehensive set of tools for software development, including debugging tools. IDEs typically include a debugger that allows developers to pause their code at specific points and examine the state of the program.
24. Debuggers: Debuggers are standalone software tools that allow developers to examine the behavior of their code at runtime. Debuggers allow developers to set breakpoints in their code, pause execution at specific points, and step through their code one line at a time.
25. Profilers: Profilers are tools that help developers identify performance bottlenecks in their code. Profilers collect data about how the code is executed and provide developers with insights into which parts of the code are taking the most time or using the most resources.
26. Memory analyzers: Memory analyzers are tools that help developers identify memory-related issues in their code, such as memory leaks or buffer overflows. Memory analyzers allow developers to examine the memory usage of their code and identify areas where memory is being used inefficiently.
27. Log analyzers: Log analyzers are tools that help developers examine the log files generated by their code. Log analyzers can help developers identify issues in their code by providing insights into how the code is behaving and how it is interacting with other parts of the system.

Debugging tools can help developers identify and fix issues in their code quickly and effectively, which can help improve the quality and performance of the software they are developing.

## Remote debugging

Remote debugging is the process of debugging a program running on a remote machine, typically over a network connection. It allows developers to debug code on a remote machine without having to physically access that machine.

Remote debugging typically involves two components: a debugging client and a debugging server. The client is the tool used by the developer to connect to the remote machine and debug the code. The server is the program running on the remote machine that allows the client to connect to it and debug the code.

To perform remote debugging, the developer typically sets up the server program on the remote machine and then connects to it using the client tool. Once connected, the developer can set breakpoints, step through code, examine variables, and perform other debugging tasks.

Remote debugging can be useful in a variety of situations, such as when the developer does not have physical access to the machine running the code, or when the code is running on a server in a remote data center. It can also be useful for debugging distributed systems, where multiple machines are involved in the processing of a task.

However, remote debugging can also be more complex than local debugging, as it requires a network connection and often involves additional security considerations. It is important for developers to carefully configure and secure the remote debugging environment to ensure the security and stability of the system.

Standard Remote Debugging:

Standard remote debugging refers to the use of a standardized protocol for remote debugging. The protocol defines the communication between the debugger and the debugging agent and ensures that they can work together seamlessly, regardless of the language or platform being used.

There are several standardized remote debugging protocols, each designed for specific languages or platforms.

Here are some examples:

GDB Remote Serial Protocol: This protocol is used for remote debugging of applications written in C, C++, and other languages that use the GNU Debugger (GDB).

Java Debug Wire Protocol (JDWP): This protocol is used for remote debugging of Java applications.

There are several standards for remote debugging that are commonly used in the software development industry. Here are some of the most popular standards:

28. GDB: GDB is a command-line debugger that is widely used in the Unix/Linux ecosystem. It supports remote debugging through the GDB remote protocol, which enables developers to debug code running on a remote system using a local instance of GDB.
29. Java Remote Debugging: Java developers often use the Java Debug Wire Protocol (JDWP) to remotely debug Java applications. This protocol allows developers to connect a Java debugger to a remote Java Virtual Machine (JVM) and inspect the application's state, set breakpoints, and step through code execution.
30. Visual Studio Remote Debugging: Microsoft Visual Studio provides a built-in remote debugging feature that allows developers to debug code running on a remote Windows system. It uses the Visual Studio Remote Debugging Monitor, which is installed on the remote system, to facilitate the debugging session.
31. Chrome DevTools Protocol: The Chrome DevTools Protocol allows developers to remotely debug and profile web applications running in the Chrome browser. This protocol enables developers to interact with the browser's rendering engine, inspect DOM elements, and monitor network activity.
32. XDebug: XDebug is a PHP debugging tool that supports remote debugging through the DBGP protocol. It enables developers to connect a local PHP debugger to a remote PHP process and debug the code running on that process.

These standards provide developers with a range of options for remotely debugging software applications, depending on their specific development environment and requirements.

## Extended Remote Debugging

Extended remote debugging refers to the process of debugging a software application that is running on a remote machine, usually over a network or the internet. It allows developers to access and analyze the code, variables, and other important information about the application without physically being on the same machine as the software.

This type of debugging is particularly useful when dealing with distributed systems, where different components of the application are running on different machines, or when the software is deployed to production environments where access is limited. Extended remote debugging is typically done using specialized debugging tools that provide features such as

Remote code execution.

Live code editing.

Real-time debugging information.

Extended remote debugging refers to the process of debugging software code running on a remote system with additional capabilities beyond the standard remote debugging features. This can include features such as recording and replaying code execution, profiling and optimizing code performance, and capturing and analyzing runtime data.

Extended remote debugging typically requires more advanced debugging tools and techniques than standard remote debugging. Here are some examples of extended remote debugging techniques:

33. Record and Replay Debugging: This technique involves recording the execution of an application on a remote system and replaying it on a local machine to identify and reproduce bugs. Tools like rr and Chronon allow developers to record and replay code execution, making it easier to pinpoint issues that are difficult to reproduce.
34. Profiling and Optimization: Profiling tools like perf, gprof, and Intel VTune allow developers to identify performance bottlenecks and optimize their code. These tools can be used remotely to profile code running on a remote system, providing developers with valuable insights into the performance of their applications.
35. Remote Debugging with Docker: Docker is a popular containerization platform that enables developers to create lightweight, portable environments for their applications. Debugging Docker containers remotely can be challenging, but tools like Delve and Visual Studio Code's remote debugging extension make it possible to debug code running inside containers.
36. Remote Debugging with Kubernetes: Kubernetes is a container orchestration platform that is widely used for deploying and managing containerized applications at scale. Remote debugging in a Kubernetes environment can be complex, but tools like Telepresence and Skaffold provide developers with tools to remotely debug code running in a Kubernetes cluster.

Extended remote debugging techniques can be invaluable for debugging complex applications running in production environments. By providing developers with additional tools and insights, extended remote debugging can help reduce downtime and improve the overall quality of software applications.

GDB Server

GDBserver is a component of the GNU Debugger (GDB) that enables remote debugging of target systems. It allows developers to debug programs running on a remote machine or embedded system without having to physically access that machine. Instead, GDBserver runs on the target system and communicates with GDB running on the development host.

To use GDBserver, the developer first starts the GDBserver program on the target system, specifying the port number and any other necessary configuration options. Then, on the development host, they start GDB and connect to the remote target by specifying the IP address and port number of the machine running.

GDBserver. GDB can then communicate with the target system and debug the program running on it as if it were running locally.

GDBserver supports a variety of communication protocols, including TCP/IP, serial port, and USB, making it flexible enough to be used in a variety of situations. It can also be used in conjunction with cross-compilers to debug programs built for different architectures than the development host.

Overall, GDBserver is a powerful tool for remote debugging that enables developers to debug programs running on embedded systems and other remote targets with ease.

When a program is being debugged using GDB Server, the GDB on the host computer sends commands to the GDB Server, which then executes those commands on the target system. The GDB Server communicates the results of those commands back to the GDB on the host computer, allowing the developer to debug the program as if it were running locally.

GDB Server is particularly useful for embedded systems development, where the target system may not have a full development environment and resources are limited. GDB Server allows developers to debug their code on the target system without the need to install a full development environment or even a full version of GDB.

GDB Server multiarch

GDB Server Multiarch is an extension to the GDB Server that enables remote debugging of multi-architecture systems, i.e., systems that have multiple CPU architectures. GDB Server Multiarch allows developers to debug programs running on a target system that has multiple CPU architectures by setting up a communication channel between the GDB on the host computer and the GDB Server running on the target system.

When debugging a multi-architecture system, the GDB Server Multiarch detects the architecture of the target system and automatically switches between different GDB Server instances that support each architecture. This allows developers to debug programs running on different architectures without the need to manually switch between different debugging sessions.

GDB Server Multiarch is particularly useful for embedded systems development, where the target system may have multiple CPUs or CPU architectures. For example, a system-on-chip (SoC) device may have both an ARM Cortex-A CPU for running Linux and an ARM Cortex-M CPU for running real-time tasks. With GDB Server Multiarch, developers can debug both CPUs simultaneously and seamlessly switch between debugging sessions for each architecture.

Some popular GDB Server Multiarch implementations include OpenOCD, which supports a wide range of CPU architectures, including ARM, MIPS, PowerPC, and RISC-V, among others. J-Link GDB Server also supports multiple architectures, including ARM, Cortex-M, and RISC-V. Segger GDB Server supports multiple architectures, including ARM, Cortex-M, and MIPS, among others.

Overall, GDB Server Multiarch is a valuable tool for developers working with multi-architecture systems, allowing them to debug programs running on different architectures simultaneously and efficiently.

Installing gdb-multiarch

The host needs a cross-debugger to debug an application running on the target. GDB (GNU Debugger)

has a version which supports multiple architectures (such as ARM, MIPS, ...) named gdb-multiarch.

1. Install GDB and GDB multi-architecture:

```
(host)$ sudo apt-get install gdb gdb-multiarch
```

2. Run gdb-multiarch and check its version.

```
(host)$ gdb-multiarch -v
```

• Remove any existing versions of GDB and GDB multi-architecture from the host:

```
(host)$ sudo apt-get purge gdb gdb-multiarch
```

On the host, in the directory of your helloWorld executable, launch the cross-debugger:

```
(host)$ gdb-multiarch (---executable file--)
```

he commands available in GDB Server Multiarch depend on the specific implementation of GDB Server being used, as well as the CPU architectures being debugged. However, there are some common GDB commands that are typically used in GDB Server Multiarch.

37. target: This command is used to specify the target system for remote debugging. For example, target remote :3333 connects to a GDB Server instance running on the target system over TCP/IP port 3333.
38. set architecture: This command is used to set the architecture of the target system. For example, set architecture armv7-m sets the target system architecture to ARM Cortex-M.
39. monitor: This command is used to send monitor commands to the GDB Server instance running on the target system. For example, monitor reset resets the target system.
40. info: This command is used to display information about the target system, such as the currently selected CPU architecture. For example, info reg displays the contents of the CPU registers.
41. step: This command is used to execute the current instruction and stop at the next instruction. For example, stepi steps a single instruction and stops.
42. continue: This command is used to continue execution of the program until the next breakpoint or other stop condition is encountered. For example, continue resumes execution of the program.
43. breakpoint: This command is used to set a breakpoint at a specific location in the code. For example, break main sets a breakpoint at the beginning of the main function.

These are just a few examples of the commands available in GDB Server Multiarch. The specific commands and syntax may vary depending on the GDB Server implementation and CPU architecture being debugged.

Features of GDB Server

Remote Debugging: gdbserver allows developers to debug applications that are running on remote machines or devices, even if the local machine doesn't have the same architecture or operating system.

Multi-Thread Support: gdbserver has multi-thread support, which allows developers to debug multi-threaded applications remotely.

Symbol Table Support: gdbserver can read symbol tables generated by compilers, which allows it to display function names and other debugging information.

Breakpoints: gdbserver allows setting and removing breakpoints, which is useful when trying to pinpoint the exact location of a bug in the code.

Signal Handling: gdbserver can handle signals such as SIGINT and SIGTERM, which are sent to the application being debugged.

Core Dump Analysis: gdbserver can analyze core dumps generated by a remote process, which can help identify the root cause of a crash.

Reverse Debugging: gdbserver can be used for reverse debugging, which means stepping backwards through code execution to identify the cause of a problem.

JTAG debugging: Many GDB Server implementations support JTAG debugging, which allows developers to debug programs running on embedded systems that have JTAG interfaces.

Flash programming: Some GDB Server implementations support flash programming, which allows developers to program the flash memory of the target system from the GDB command line.

Trace debugging: Some GDB Server implementations support trace debugging, which allows developers to capture and analyze trace data from the target system.

Customization: GDB Server can be customized to support specific target systems and CPU architectures, and can be integrated into development environments such as Eclipse and Visual Studio.

Cross-platform support: GDB Server supports debugging of programs running on a wide range of platforms, including Linux, Windows, macOS, and various embedded operating systems.

Rules for GDB server-client communication

GDB (GNU Debugger) is a powerful tool used for debugging and analyzing software applications. gdbserver is a component of GDB that allows remote debugging of applications running on a target system. When using gdbserver for remote debugging, there are several rules that should be followed to ensure effective communication between the gdbserver and client components.

Compatibility: The gdbserver and client versions should be compatible with each other. If they are not compatible, it may result in errors during communication or unexpected behavior.    Protocol: gdbserver and the client should use the same protocol for communication. The protocol can be selected using the "-x" command line option with gdbserver.

Network connectivity: gdbserver and the client should be connected to the same network or reachable through a secure channel. If there is a firewall or other network security measures, they should be configured to allow communication between the gdbserver and client.

Permissions: The user running gdbserver should have sufficient permissions to execute and debug the application being debugged. Additionally, any necessary permissions should be granted to the client machine to connect and communicate with the gdbserver.

Debugging Symbols: The application being debugged should be compiled with debugging symbols (-g option), so that gdbserver can provide the necessary information to the client to perform.

The GDB server-client communication protocol follows a set of rules to ensure that the debugging session proceeds smoothly and efficiently. Here are some of the key rules for GDB server-client communication:

44. Protocol compatibility: The GDB client and server must use the same communication protocol version to ensure compatibility. If the protocol version does not match, the GDB client may not be able to communicate with the GDB server.
45. Connection setup: The GDB client must establish a connection with the GDB server using a supported communication protocol, such as TCP/IP or serial.
46. Target system identification: The GDB server must identify the target system and its architecture to the GDB client, using the target command.
47. Memory access: The GDB client must use the read and write commands to access the memory of the target system. The GDB server is responsible for handling the requests and ensuring that the memory access is valid.
48. Breakpoint management: The GDB client must use the breakpoint command to set breakpoints in the target program. The GDB server is responsible for handling the breakpoints and suspending the program execution when a breakpoint is hit.
49. Signal handling: The GDB client and server must communicate effectively when handling signals sent to the target program. The GDB server is responsible for notifying the GDB client when a

signal is received by the target program, and the GDB client can use the signal command to handle the signal.

50. Program execution control: The GDB client can use commands such as step, next, and continue to control the execution of the target program. The GDB server is responsible for executing the program and communicating its status to the GDB client.

Overall, the rules for GDB server-client communication are designed to ensure that the debugging session is reliable, efficient, and effective. Developers must be familiar with these rules to use GDB effectively and debug their programs efficiently.

Special commands used for GDBserver

target remote <host>:<port>: This command connects GDB to the remote GDBserver running on the specified host and port.

set remotebaud <baudrate>: This command sets the baud rate for serial connections. //Serial communication

monitor reset: This command resets the target system.

monitor exit: This command exits GDBserver.

monitor help: This command displays a list of available monitor commands.

monitor resume: This command resumes the execution of the target system.

51. monitor: This command is used to send commands directly to the GDBserver. For example, the command monitor reset can be used to reset the target system.
52. kill: This command is used to terminate the debugging session.
53. detach: This command is used to detach the GDB client from the GDBserver, leaving the target program running.
54. file: This command is used to specify the executable file to be debugged.
55. set: This command is used to set various GDBserver options, such as the remote protocol, communication port, and target architecture.
56. show: This command is used to display various GDBserver options and settings.
57. remote: This command is used to control the remote communication protocol, such as setting the remote TCP/IP or serial port.

58. reset: This command is used to reset the target system, including its CPU and peripherals.
59. poll: This command is used to monitor the target system and detect when it stops responding.
60. load: This command is used to load the executable file onto the target system's memory.

These are just a few examples of the special commands that are available in GDBserver. Depending on the specific implementation of GDBserver that you are using, there may be additional commands or variations of these commands available. It is important to consult the documentation for your specific GDBserver implementation to learn about all the available commands and how to use them.

Copy the file

Using remote get command in gdb: You can use the file get command in gdb to copy the executable file from the remote machine where gdbserver is running to the local machine where gdbclient is running.

```
  remote get <path of remote file> <space> < local path >
```

Using SCP: SCP (Secure Copy) is a command-line utility that allows you to securely copy files between remote and local machines. You can use SCP to copy the executable file from the remote machine to the local machine. Here's an example command to copy the file:

```
scp user@remote:/path/to/remote/executable /path/to/local/destination
```

where user is the username on the remote machine, remote is the IP address or hostname of the remote machine, /path/to/remote/executable is the path to the executable file on the remote machine, and /path/to/local/destination is the path where you want to save the file on the local machine.

File Transfer Protocol:

File Transfer Protocol (FTP) is a standard network protocol used to transfer files between clients and servers on a

computer network. The following are some of the commonly used FTP commands:

USER: This command is used to send the username to the FTP server.

PASS: This command is used to send the password to the FTP server.

LIST: This command is used to list the files and directories on the remote server.

RETR: This command is used to retrieve a file from the remote server.

STOR: This command is used to store a file on the remote server.

DELE: This command is used to delete a file on the remote server.

CWD: This command is used to change the current working directory on the remote server.

PWD: This command is used to display the current working directory on the remote server.

SIZE: This command is used to get the size of a file on the remote server.

ABOR: This command is used to abort the current file transfer operation.

QUIT: This command is used to terminate the FTP session.

Implementation FTP

Using FTP: FTP (File Transfer Protocol) is a standard protocol used for transferring files over a network.

You can use an FTP client to transfer the executable file from the remote machine to the local machine.

You'll need to have an FTP server running on the remote machine and an FTP client installed on the local machine. Here's an example command to copy the file using the ftp command:

```
ftp remote
ftp> get /path/to/remote/executable /path/to/local/destination
```

where remote is the IP address or hostname of the remote machine, /path/to/remote/executable is the path to the executable file on the remote machine, and /path/to/local/destination is the path where you want to save the file on the local machine.

SFTP (Secure File Transfer Protocol):

Using SFTP: SFTP (Secure File Transfer Protocol) is a secure version of FTP that encrypts data during transfer.

You can use an SFTP client to transfer the executable file from the remote machine to the local machine. You'll need to have an SFTP server running on the remote machine and an SFTP client installed on the local machine.

Here's an example command to copy the file using the sftp command:

```
sql sftp user@remote

sftp> get /path/to/remote/executable /path/to/local/destination
```

where user is the username on the remote machine, remote is the IP address or hostname of the remote machine,

/path/to/remote/executable is the path to the executable file on the remote machine.

/path/to/local/destination is the path where you want to save the file on the local machine.

Remote debugging is a process of identifying and fixing errors in software code on a device that is not physically present with you. The following are the general steps for remote debugging:

61. Connect to the remote device: Establish a connection with the device you want to debug. You can use tools like SSH, remote desktop, or a virtual private network (VPN) to connect to the remote device.
62. Set up a debugging environment: Install the necessary debugging tools and software on the remote device. This can include debuggers, loggers, and performance monitoring tools.
63. Start the debugging process: Run the code that you want to debug on the remote device, and start the debugging process using the debugging tools you've installed.
64. Identify the problem: Use the debugging tools to identify the problem in the code. This can involve stepping through the code line by line, setting breakpoints, and examining variable values and stack traces.
65. Fix the problem: Once you have identified the problem, make the necessary changes to the code to fix the issue. You may need to modify the code directly on the remote device or upload a new version of the code.
66. Test the code: After making changes, test the code on the remote device to ensure that the issue has been resolved.
67. Disconnect from the remote device: Once you have finished remote debugging, disconnect from the remote device and ensure that all debugging tools and software have been removed.

Note that the specific steps for remote debugging can vary depending on the programming language, platform, and tools used.

Here are some commonly used commands for remote debugging with gdbserver:

68. target remote <hostname>:<port>: This command connects to the remote gdbserver running on the specified host and port.
69. file <executable>: This command loads the specified executable file into gdb on the remote machine.
70. break <function_name>: This sets a breakpoint at the specified function name in the executable.
71. continue: This command resumes execution of the program being debugged.
72. step: This command steps the program forward one line of code at a time.
73. next: This command steps over the next line of code without entering any functions.
74. info locals: This command displays the values of all local variables at the current point in the program.
75. info breakpoints: This command displays information about all currently set breakpoints.
76. delete <breakpoint_number>: This command removes the specified breakpoint.
77. detach: This command detaches gdb from the remote program, leaving it running.
78. quit: This command terminates gdb and ends the debugging session.

These are just a few examples of commands you might use when remote debugging with gdbserver. The full set of available commands can be found in the gdb manual.

## 1. Install gbdserver on Target System

Target machine is the one which is running the program which you have to debug. You need to have the "gdbserver" executable on the target machine.

```
$ sudo apt-get install gdbserver
```

To do remote debugging, start your program using the gdbserver. gdbserver then automatically suspends the execution of your program at its entry point, and it waits for a debugger to connect to it. gdbserver doesn't need the symbols from your program to debug. So you can strip symbols out of your program binary to save space.

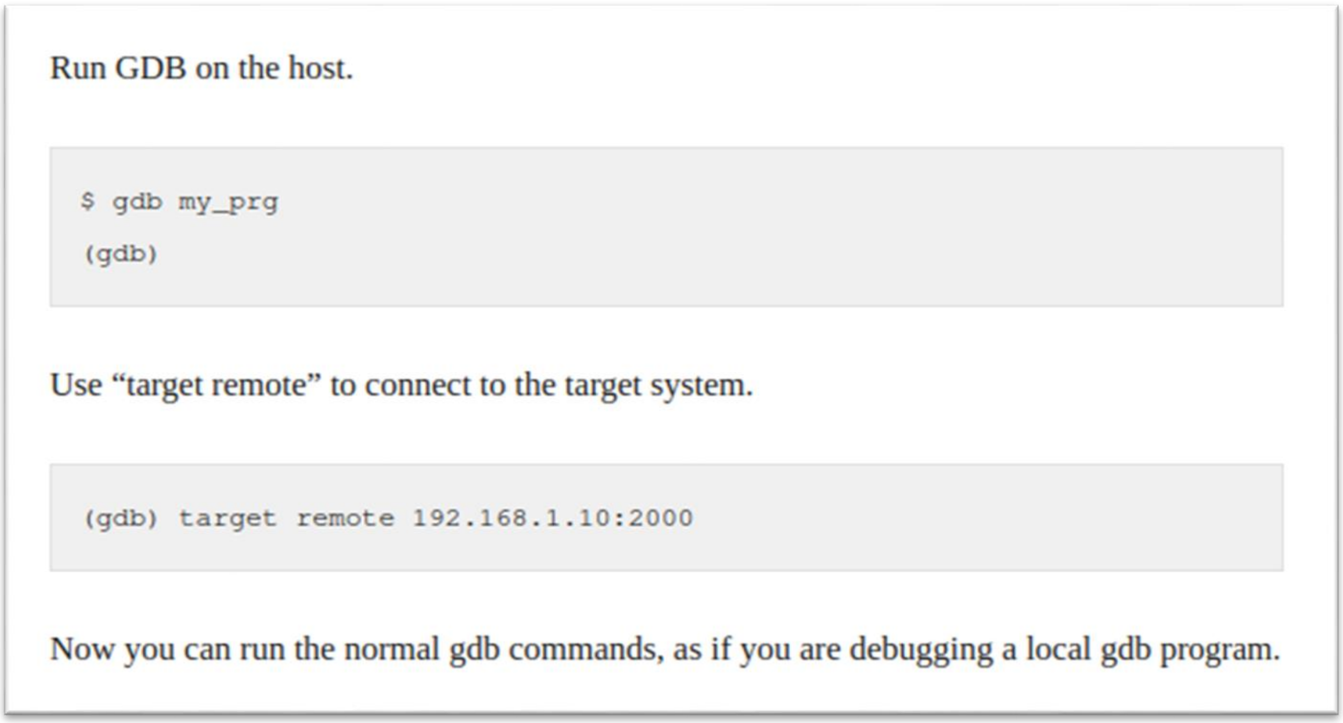
```
$ gdbserver localhost:2000 my_prg


Process program created; pid = 2045
Listening on port 2000
```

The above command suspend the execution on my_prg, and waits for a debugger to connect to it on port 2000.

## 2. Launch gdb on Host System

The executable file and the libraries in the host, must exactly match the executable file and libraries on the target, with an exception that the target binary symbols can be stripped. You can also load the symbols separately in the host using "file" command in gdb.


Run GDB on the host.

```
$ gdb my_prg
(gdb)
```

Use “target remote” to connect to the target system.

```
(gdb) target remote 192.168.1.10:2000
```

Now you can run the normal gdb commands, as if you are debugging a local gdb program.

## 3. Remote Debugging Demo Example

The following C program example will be used to demonstrate the remote debugging.

```
#include <stdio.h>

int power(int,int);

int main() {

        int i;
        printf("Program to calculate power\n");
        for (i=0;i<10;i++)
                printf("%d %d\n",i, power(2,i));
        return 0;
}

int power (int base, int n) {

        int i,p;
        p=1;
        for (i=1; i<=n; i++)
                p = p*base;
        return p;
}

$ cc -g -o my_prg power.c
```

On Target Machine,

```
$ gdbserver localhost:2000 my_prg
Process my_prg created; pid = 20624
Listening on port 2000
```

On Host Machine,

```
$ gdb my_prg

(gdb) target remote 192.168.1.10:2000
Remote debugging using 192.168.1.10:2000
Loaded symbols for /lib64/ld-linux-x86-64.so.2
0x00007ffff7dddaf0 in ?? () from /lib64/ld-linux-x86-64.so.2
(gdb) b main
Breakpoint 1 at 0x400550
(gdb) continue
Continuing.

Breakpoint 1, 0x0000000000400550 in main ()
```

Now we have connected the gdb for remote debugging. In the last example, we have put a breakpoint in main() function. If we continue our program, the output of the program will be printed in the target machine.

On Host:

```
(gdb) continue
```

On Target:

```
Remote debugging from host 192.168.1.20
Program to calculate power
0 1
1 2
2 4
3 8
4 16
5 32
6 64
7 128
8 256
9 512

Child exited with status 0
GDBserver exiting
```

## 4. Attach gdb to a Running Process on Target

First you have to find the process ID of the running process in target.

On Host,

```
(gdb) attach 3850
```

Now the gdb will suspend the process 3850 in the target and you can debug the program using normal gdb commands.

From the above snippet,

1. ‘target extended-remote’ is used to run gdbserver in multi process mode.
2. ‘set remote exec-file /my_prg’ is used to set the program which you want to debug in the target.
3. ‘file /my_prg’ is used to load the debugging symbols from the program in the host.
4. ‘b main’ is used to set breakpoint at main() function.
5. ‘run’ is used to run the program, which stops at the breakpoint main().

Note: In the above case, the executable “my_prg” is present under “/” on both target and host.

Now either you can ‘continue’ or ‘detach’ the program from debugging. Still the gdbserver will not exit in the target machine, so you can change the ‘remote exec-file’ at any time, and debug a different set of program.

Flow chart

79. Set up the remote environment
80. Connect to the remote system
81. Start the debugging session
82. Run the code remotely
83. Set breakpoints in the code
84. Debug the code remotely
85. Monitor code execution remotely
86. Repeat steps 5-7 as necessary to identify and fix bugs
87. End the debugging session
88. Disconnect from the remote system

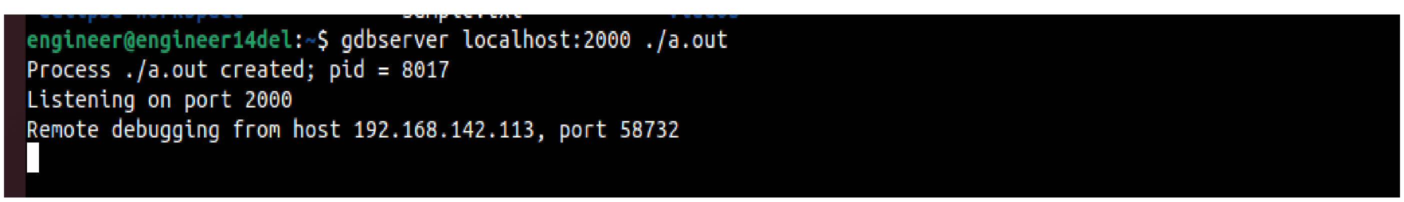
This flow chart shows the basic steps involved in using gdbserver for remote debugging. The local host runs gdb, which connects to a gdbserver instance running on the target system. Once connected, gdb can be used to set breakpoints, examine variables, and debug the code running on the target system. When debugging is complete, gdb and gdbserver are both shut down, and the connection between the local host and target system is terminated.

```
 +------------+           +-----------------------+
 | Local host |           | Remote system (target)
|
 +------------+           +-----------------------+
       |                                     |
       |          SSH or other               |
       |          remote access              |
       |------------------------------------->|
       |                                     |
       |       Start gdbserver               |
       |------------------------------------->|
       |                                     |
       | Connect to gdbserver on target |
       |<-------------------------------------|
       |                        |
       |      Launch gdb on local      |
       |                        |
       |------------------------------------->|
       |                        |
       |  Set breakpoints, examine     |
       |  variables, step through code |
       |<-------------------------------------|
       |                        |
       |    Monitor code execution     |
       |                        |
       |-------------------------------------->|




       |                    |
       |     Debug the code        |
       |                    |
       |-------------------------------------->|
       |                    |
       |   Repeat debugging steps    |
       |<--------------------------------------|
       |                    |
       |   End gdb and gdbserver      |
       |                    |
       |-------------------------------------->|
       |                    |
       |  Disconnect from the target  |
       |<--------------------------------------|
       |                    |
    +------------+          +-----------------------+
    | Local host |          | Remote system (target)  |
    +------------+          +-----------------------+
```

## Successful connection of client



```
engineer@engineer14del:~$ gdbserver localhost:2000 ./a.out
Process ./a.out created; pid = 8017
Listening on port 2000
Remote debugging from host 192.168.142.113, port 58732
```

## Building the Kernel Locally

On a Raspberry Pi, first install the latest version of Raspberry Pi OS. Then boot your Raspberry Pi, log in, and ensure you're connected to the internet to give you access to the sources.

First install Git and the build dependencies:

```
sudo apt install git bc bison flex libssl-dev make
```

Next get the sources, which will take some time:

```
git clone --depth=1 https://github.com/raspberrypi/linux
```

### *Choosing Sources*

The git clone command above will download the current active branch (the one we are building Raspberry Pi OS images from) without any history. Omitting the --depth=1 will download the entire repository, including the full history of all branches, but this takes much longer and occupies much more storage.

To download a different branch (again with no history), use the --branch option:

```
git clone --depth=1 --branch <branch> https://github.com/raspberrypi/linux
```

where <branch> is the name of the branch that you wish to download.

Refer to the original GitHub repository for information about the available branches.

### *Kernel Configuration*

Configure the kernel; as well as the default configuration, you may wish to configure your kernel in more detail or apply patches from another source, to add or remove required functionality.

#### Apply the Default Configuration

First, prepare the default configuration by running the following commands, depending on your Raspberry Pi model:

**For Raspberry Pi 1, Zero and Zero W, and Raspberry Pi Compute Module 1 default (32-bit only) build configuration**

```
cd linux
KERNEL=kernel
make bcmrpi_defconfig
```

**For Raspberry Pi 2, 3, 3+ and Zero 2 W, and Raspberry Pi Compute Modules 3 and 3+ default 32-bit build configuration**

```
cd linux
KERNEL=kernel7
make bcm2709_defconfig
```

**For Raspberry Pi 4 and 400, and Raspberry Pi Compute Module 4 default 32-bit build configuration**

```
cd linux
KERNEL=kernel7l
make bcm2711_defconfig
```

**For Raspberry Pi 3, 3+, 4, 400 and Zero 2 W, and Raspberry Pi Compute Modules 3, 3+ and 4 default 64-bit build configuration**

```
cd linux
KERNEL=kernel8
make bcm2711_defconfig
```

## Customising the Kernel Version Using LOCALVERSION

In addition to your kernel configuration changes, you may wish to adjust the LOCALVERSION to ensure your new kernel does not receive the same version string as the upstream kernel. This both clarifies you are running your own kernel in the output of uname and ensures existing modules in /lib/modules are not overwritten.

To do so, change the following line in .config:

```
CONFIG_LOCALVERSION="-v7l-MY_CUSTOM_KERNEL"
```

You can also change that setting graphically as shown in the kernel configuration instructions. It is located in "General setup" => "Local version - append to kernel release".

## *Building the Kernel*

Build and install the kernel, modules, and Device Tree blobs; this step can take a **long** time depending on the Raspberry Pi model in use.

**For the 32-bit kernel:**

```
make -j4 zImage modules dtbs
sudo make modules_install
sudo cp arch/arm/boot/dts/*.dtb /boot/
sudo cp arch/arm/boot/dts/overlays/*.dtb* /boot/overlays/
sudo cp arch/arm/boot/dts/overlays/README /boot/overlays/
sudo cp arch/arm/boot/zImage /boot/$KERNEL.img
```

**For the 64-bit kernel:**

```
make -j4 Image.gz modules dtbs
sudo make modules_install
sudo cp arch/arm64/boot/dts/broadcom/*.dtb /boot/
sudo cp arch/arm64/boot/dts/overlays/*.dtb* /boot/overlays/
sudo cp arch/arm64/boot/dts/overlays/README /boot/overlays/
sudo cp arch/arm64/boot/Image.gz /boot/$KERNEL.img
```

| | |
|---|---|
| Note | **On a Raspberry Pi 2/3/4, the -j4 flag splits the work between all four cores, speeding up compilation significantly.** |

If you now reboot, your Raspberry Pi should be running your freshly-compiled kernel!

## Cross-Compiling the Kernel

First, you will need a suitable Linux cross-compilation host. We tend to use Ubuntu; since Raspberry Pi OS is also a Debian distribution, it means many aspects are similar, such as the command lines.

You can either do this using VirtualBox (or VMWare) on Windows, or install it directly onto your computer. For reference, you can follow instructions online at Wikihow.

### *Install Required Dependencies and Toolchain*

To build the sources for cross-compilation, make sure you have the dependencies needed on your machine by executing:

```
sudo apt install git bc bison flex libssl-dev make libc6-dev libncurses5-dev
```

If you find you need other things, please submit a pull request to change the documentation.

## Install the 32-bit Toolchain for a 32-bit Kernel

```
sudo apt install crossbuild-essential-armhf
```

## Install the 64-bit Toolchain for a 64-bit Kernel

```
sudo apt install crossbuild-essential-arm64
```

### *Get the Kernel Sources*

To download the minimal source tree for the current branch, run:

```
git clone --depth=1 https://github.com/raspberrypi/linux
```

See [Choosing sources]() above for instructions on how to choose a different branch.

## *Build sources*

Enter the following commands to build the sources and Device Tree files:

### 32-bit Configs

**For Raspberry Pi 1, Zero and Zero W, and Raspberry Pi Compute Module 1:**

```
cd                                                                    linux
KERNEL=kernel
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- bcmrpi_defconfig
```

**For Raspberry Pi 2, 3, 3+ and Zero 2 W, and Raspberry Pi Compute Modules 3 and 3+:**

```
cd                                                                    linux
KERNEL=kernel7
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- bcm2709_defconfig
```

**For Raspberry Pi 4 and 400, and Raspberry Pi Compute Module 4:**

*cd linux*
*KERNEL=kernel7l*
*make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- bcm2711_defconfig*

### 64-bit Configs

**For Raspberry Pi 3, 3+, 4, 400 and Zero 2 W, and Raspberry Pi Compute Modules 3, 3+ and 4:**

*cd linux*
*KERNEL=kernel8*
*make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- bcm2711_defconfig*

### Build with Configs

| | |
|---|---|
| Note | **To speed up compilation on multiprocessor systems, and get some improvement on single processor ones, use -j n, where n is the number of processors * 1.5. You can use the nproc command to see how many processors you have. Alternatively, feel free to experiment and see what works!** |

### For all 32-bit Builds

make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- zImage modules dtbs

### For all 64-bit Builds

| | |
|---|---|
| Note | **The difference between Image target between 32 and 64-bit.** |

make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- Image modules dtbs .

### *Install Directly onto the SD Card*

Having built the kernel, you need to copy it onto your Raspberry Pi and install the modules; this is best done directly using an SD card reader.

First, use lsblk before and after plugging in your SD card to identify it. You should end up with something a lot like this:

**sdb**
**sdb1**
**sdb2**

with sdb1 being the FAT filesystem (boot) partition, and sdb2 being the ext4 filesystem (root) partition.

Mount these first, adjusting the partition letter as necessary:

```
mkdir mnt
mkdir mnt/fat32
mkdir mnt/ext4
sudo mount /dev/sdb1 mnt/fat32
sudo mount /dev/sdb2 mnt/ext4
```

| | |
|---|---|
| Note | **You should adjust the drive letter appropriately for your setup, e.g. if your SD card appears as /dev/sdc instead of /dev/sdb.** |

Next, install the kernel modules onto the SD card:

**For 32-bit**

```
sudo env PATH=$PATH make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- INSTALL_MOD_PATH=mnt/ext4 modules_install
```

**For 64-bit**

```
sudo env PATH=$PATH make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- INSTALL_MOD_PATH=mnt/ext4 modules_install
```

Finally, copy the kernel and Device Tree blobs onto the SD card, making sure to back up your old kernel:

**For 32-bit**

```
sudo cp mnt/fat32/$KERNEL.img mnt/fat32/$KERNEL-backup.img
sudo cp arch/arm/boot/zImage mnt/fat32/$KERNEL.img
sudo cp arch/arm/boot/dts/*.dtb mnt/fat32/
sudo cp arch/arm/boot/dts/overlays/*.dtb* mnt/fat32/overlays/
sudo cp arch/arm/boot/dts/overlays/README mnt/fat32/overlays/
sudo umount mnt/fat32
sudo umount mnt/ext4
```

**For 64-bit**

```
sudo cp mnt/fat32/$KERNEL.img mnt/fat32/$KERNEL-backup.img
sudo cp arch/arm64/boot/Image mnt/fat32/$KERNEL.img
sudo cp arch/arm64/boot/dts/broadcom/*.dtb mnt/fat32/
sudo cp arch/arm64/boot/dts/overlays/*.dtb* mnt/fat32/overlays/
sudo cp arch/arm64/boot/dts/overlays/README mnt/fat32/overlays/
sudo umount mnt/fat32
sudo umount mnt/ext4
```

Another option is to copy the kernel into the same place, but with a different filename - for instance, kernel-myconfig.img - rather than overwriting the kernel.img file. You can then edit the config.txt file to select the kernel that the Raspberry Pi will boot:

```
kernel=kernel-myconfig.img
```

This has the advantage of keeping your custom kernel separate from the stock kernel image managed by the system and any automatic update tools, and allowing you to easily revert to a stock kernel in the event that your kernel cannot boot.

Finally, plug the card into the Raspberry Pi and boot it!

## Configuring the Kernel

Edit this on GitHub

The Linux kernel is highly configurable; advanced users may wish to modify the default configuration to customise it to their needs, such as enabling a new or experimental network protocol, or enabling support for new hardware.

Configuration is most commonly done through the make menuconfig interface. Alternatively, you can modify your .config file manually, but this can be more difficult for new users.

### Preparing to Configure

The menuconfig tool requires the ncurses development headers to compile properly. These can be installed with the following command:

```
sudo apt install libncurses5-dev
```

You'll also need to download and prepare your kernel sources, as described in the build guide. In particular, ensure you have installed the default configuration.

### Using menuconfig

Once you've got everything set up and ready to go, you can compile and run the menuconfig utility as follows:

```
make menuconfig
```

**If you're cross-compiling a 32-bit kernel:**

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- menuconfig
```

**Or, if you are cross-compiling a 64-bit kernel:**

```
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- menuconfig
```

The menuconfig utility has simple keyboard navigation. After a brief compilation, you'll be presented with a list of submenus containing all the options you can configure; there's a lot, so take your time to read through them and get acquainted.

Use the arrow keys to navigate, the Enter key to enter a submenu (indicated by --->), Escape twice to go up a level or exit, and the space bar to cycle the state of an option. Some options have multiple choices, in which case they'll appear as a submenu and the Enter key will select an option. You can press h on most entries to get help about that specific option or menu.

Resist the temptation to enable or disable a lot of things on your first attempt; it's relatively easy to break your configuration, so start small and get comfortable with the configuration and build process.

#### Saving your Changes

Once you're done making the changes you want, press Escape until you're prompted to save your new configuration. By default, this will save to the .config file. You can save and load configurations by copying this file around.

## Patching the Kernel

Edit this on GitHub

When building your custom kernel you may wish to apply patches, or collections of patches ('patchsets'), to the Linux kernel.

Patchsets are often provided with newer hardware as a temporary measure, before the patches are applied to the upstream Linux kernel ('mainline') and then propagated down to the Raspberry Pi kernel sources. However, patchsets for other purposes exist, for instance to enable a fully pre-emptible kernel for real-time usage.

### Version Identification

It's important to check what version of the kernel you have when downloading and applying patches. In a kernel source directory, running head Makefile -n 3 will show you the version the sources relate to:

```
VERSION = 3
PATCHLEVEL = 10
SUBLEVEL = 25
```

In this instance, the sources are for a 3.10.25 kernel. You can see what version you're running on your system with the uname -r command.

### Applying Patches

How you apply patches depends on the format in which the patches are made available. Most patches are a single file, and applied with the patch utility. For example, let's download and patch our example kernel version with the real-time kernel patches:

```
wget
https://www.kernel.org/pub/linux/kernel/projects/rt/3.10/older/patch-3.10.25-rt23.patch.gz
```

```
gunzip patch-3.10.25-rt23.patch.gz
cat patch-3.10.25-rt23.patch | patch -p1
```

In our example we simply download the file, uncompress it, and then pass it to the patch utility using the cat tool and a Unix pipe.

Some patchsets come as mailbox-format patchsets, arranged as a folder of patch files. We can use Git to apply these patches to our kernel, but first we must configure Git to let it know who we are when we make these changes:

```
git config --global user.name "Your name"
git config --global user.email "your email in here"
```

Once we've done this we can apply the patches:

```
git am -3 /path/to/patches/*
```

If in doubt, consult with the distributor of the patches, who should tell you how to apply them. Some patchsets will require a specific commit to patch against; follow the details provided by the patch distributor.

## Kernel Headers

Edit this on GitHub

If you are compiling a kernel module or similar, you will need the Linux Kernel headers. These provide the various function and structure definitions required when compiling code that interfaces with the kernel.

If you have cloned the entire kernel from github, the headers are already included in the source tree. If you don't need all the extra files, it is possible to install only the kernel headers from the Raspberry Pi OS repo.

```
sudo apt install raspberrypi-kernel-headers
```

| | |
|---|---|
| Note | **It can take quite a while for this command to complete, as it installs a lot of small files. There is no progress indicator.** |

**Buildroot:**

- ➢ Buildroot is a popular open-source tool that is used to build custom embedded Linux systems. It is a software build system that automates the process of building a Linux distribution from source code. Buildroot provides a set of scripts and configuration files that make it easy to customize the Linux kernel and build system components such as libraries, applications, and device drivers.
- ➢ With Buildroot, developers can create custom embedded Linux systems for a variety of architectures including ARM, MIPS, PowerPC, and x86. The tool provides a simple way to generate a root filesystem that contains the necessary components for running a Linux system on a target device.
- ➢ Buildroot allows developers to customize various aspects of their Linux system, including the kernel configuration, the boot loader, and the root filesystem. It provides a menu-driven configuration system that makes it easy to select the components that should be included in the final image. Additionally, Buildroot can automatically download and build the required packages and dependencies, which can save developers a lot of time.
- ➢ Overall, Buildroot is a powerful tool for building custom embedded Linux systems. It is widely used in the embedded systems industry and has a large community of developers and users who contribute to its ongoing development and improvement.

- ❖ **Buildroot** is a tool that allows you to create a custom Linux system image for embedded systems. Here are the basic steps to create a Buildroot image:

- ❖ **Install Buildroot:** First, you need to download and install Buildroot on your system.

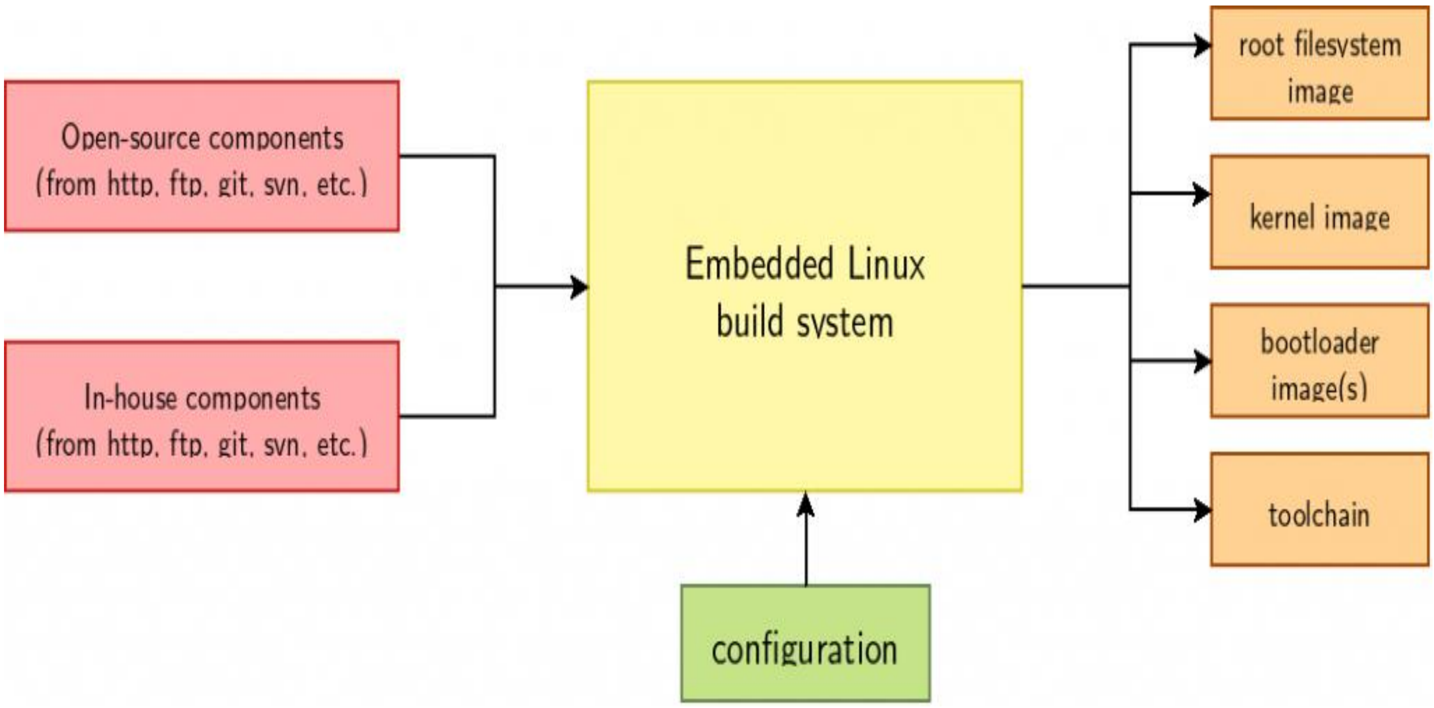
- **Configure Buildroot:** Next, you need to configure Buildroot to create the image according to your requirements. You can do  this by running the command make menuconfig.

- **Choose the target system:** Select the target architecture and system that you want to build the image for. This includes selecting the processor type, board, and kernel version.

- **Select packages**: Choose the packages you want to include in the image. This includes the libraries, utilities, and applications that you need for your system.
- **Build the image:** Once you have configured Buildroot, you can build the image by running the command make.

- **Flash the image**: After building the image, you need to flash it onto your target device. This involves copying the image file onto a bootable device, such as an SD card or USBdrive, and then inserting it into the device.

- **Test the image**: Finally, you can test the image to ensure that everything is working correctly. This involves booting up the device and verifying that the software and hardware are working as expected.



**Steps to install GDB as part of kernel image via buildroot:**

To install GDB as part of the kernel image via Buildroot, you can follow these steps:

- Add the GDB package to your Buildroot configuration by running **make menuconfig or make xconfig. Navigate to Target packages -> Debugging, profiling and benchmark -> gdb and select it.**

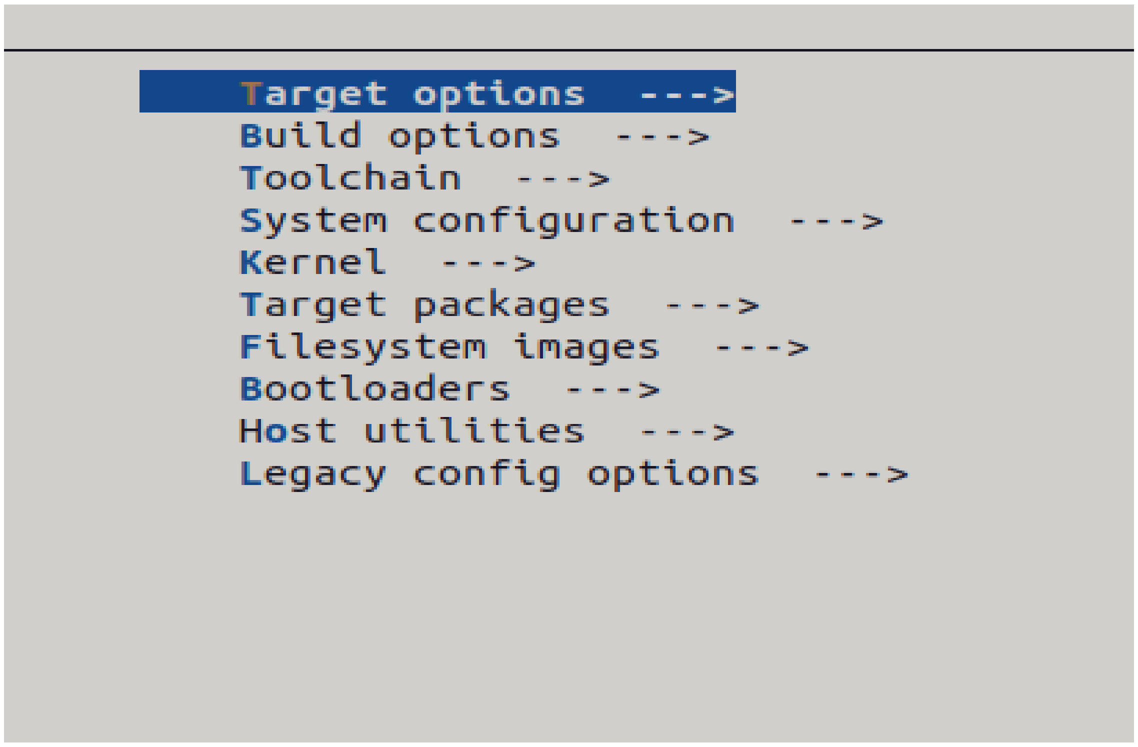
- Configure the Buildroot system by running **make menuconfig or make xconfig. Navigate to Kernel -> Kernel configuration and enable Kernel debugging and Compile the kernel with debug info. Save and exit.**
- Build the kernel image and the GDB package by running make.
- Once the build is complete, you can find the GDB binary in the output/host/bin directory.
- To add the GDB binary to the kernel image, you need to modify the kernel Makefile. Navigate to the kernel source directory and open the Makefile.
- Add the following line at the end of the file:

***makefile***

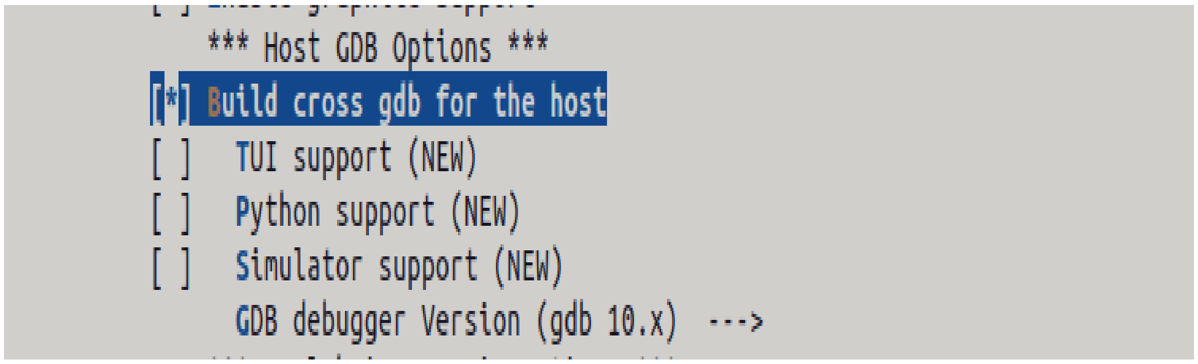
- ***KBUILD_IMAGE += $(OUTPUT_DIR)/host/bin/gdb***

Replace $(OUTPUT_DIR) with the path to your Buildroot output directory.

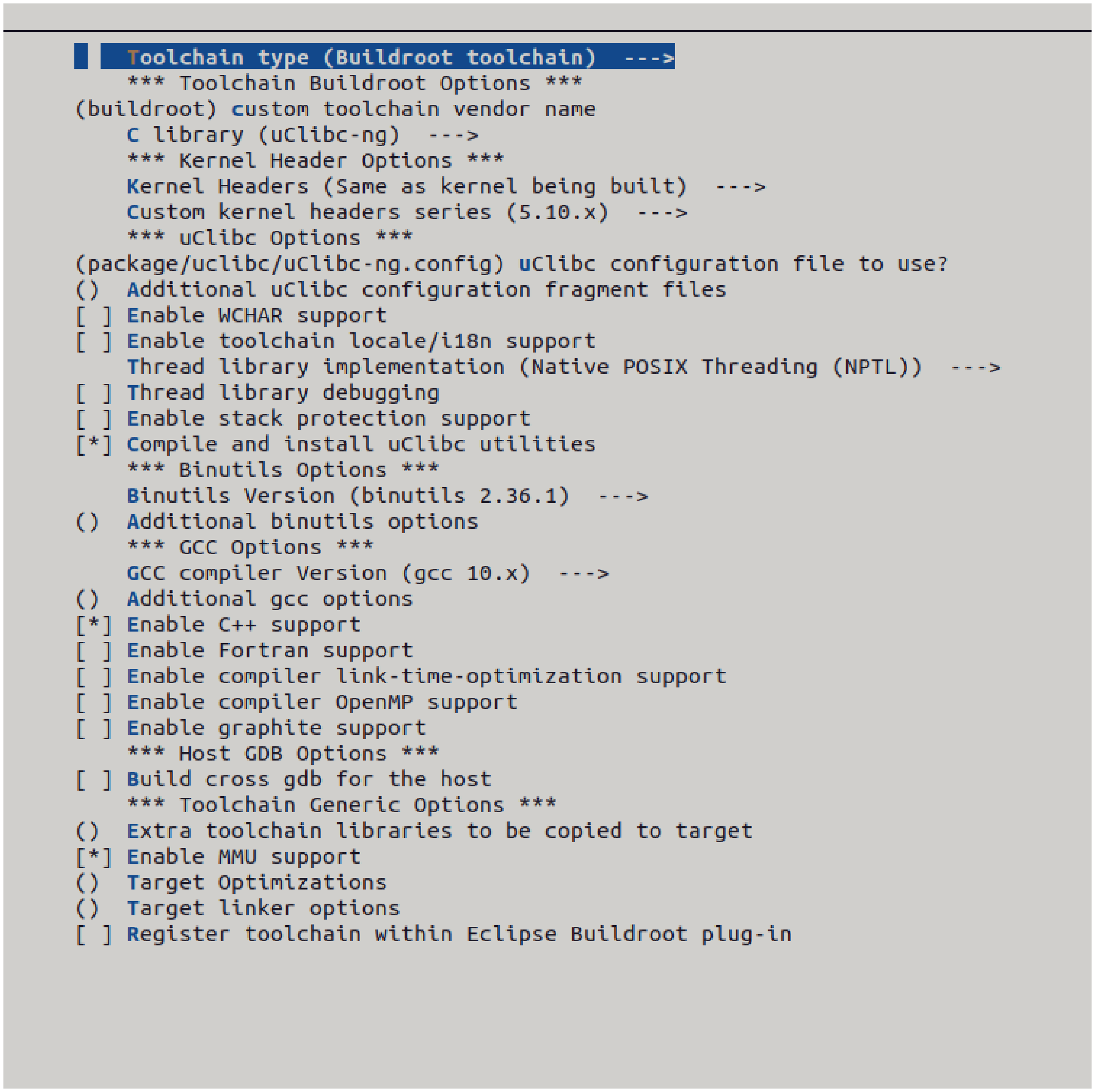
- Save the Makefile and rebuild the kernel image by running make.
- The GDB binary should now be included in the kernel image. You can verify this by running file <path-to-kernel-image> and checking that the output includes the GDB binary.



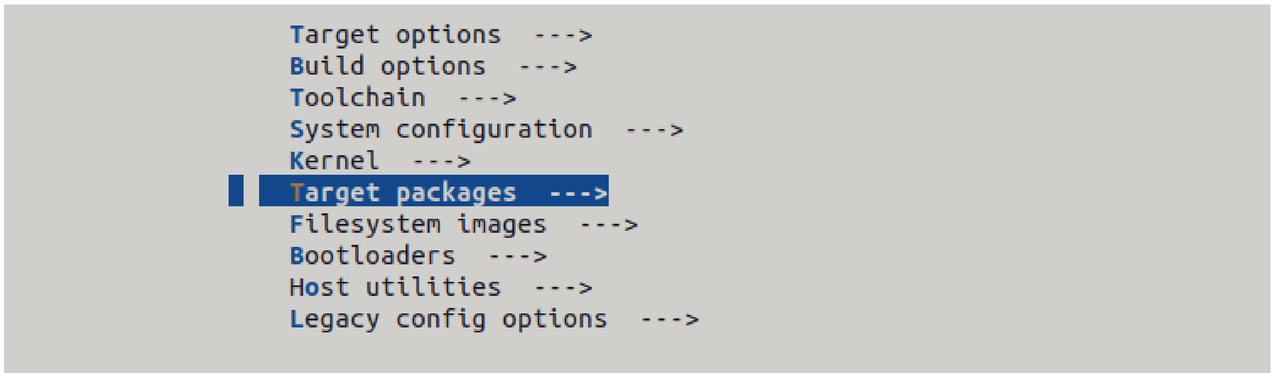
```
        Target options  --->
        Build options  --->
        Toolchain  --->
        System configuration  --->
        Kernel  --->
        Target packages  --->
        Filesystem images  --->
        Bootloaders  --->
        Host utilities  --->
        Legacy config options  --->
```



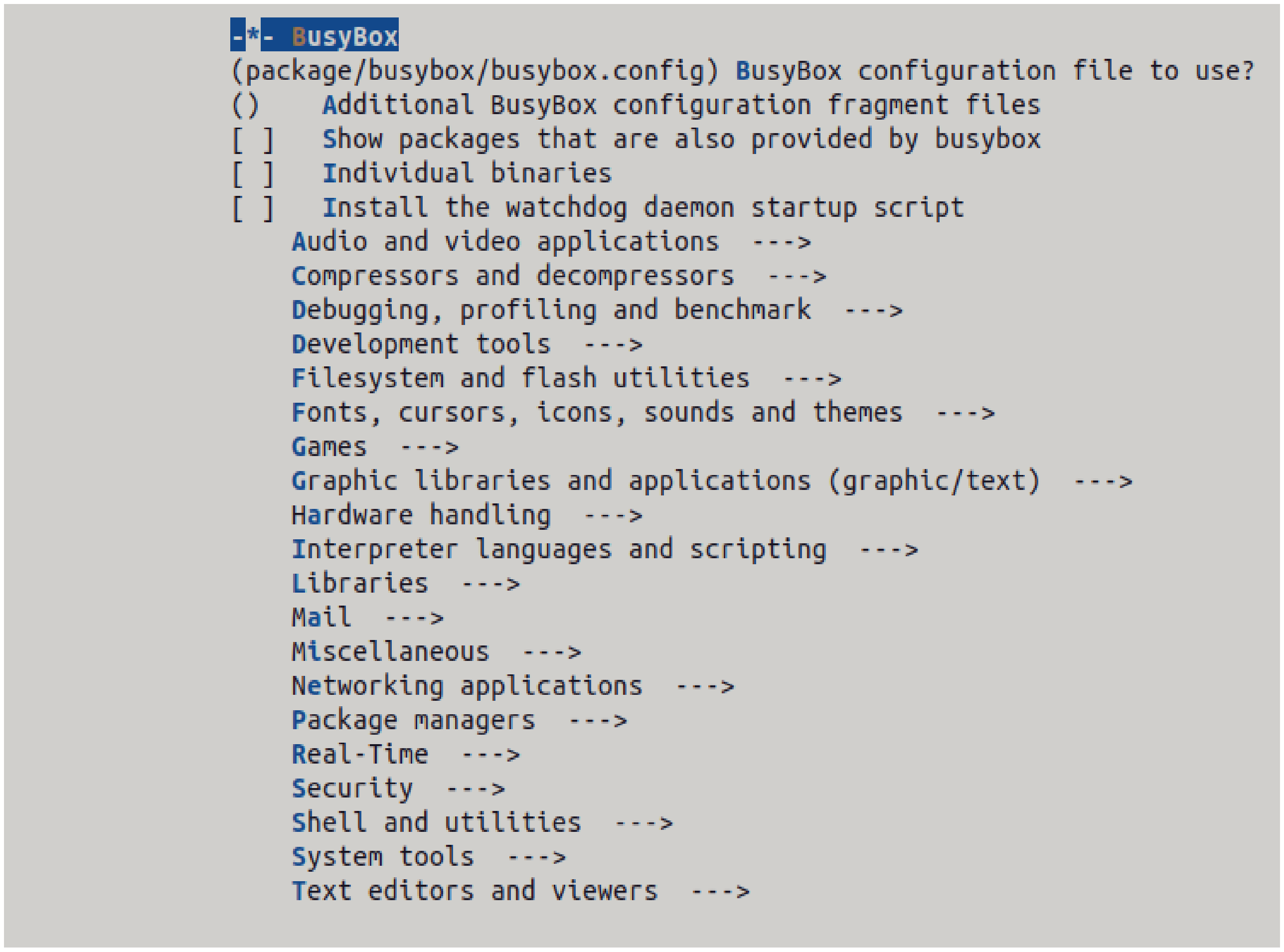
```
    *** Host GDB Options ***
[*] Build cross gdb for the host
[ ]   TUI support (NEW)
[ ]   Python support (NEW)
[ ]   Simulator support (NEW)
      GDB debugger Version (gdb 10.x)  --->
```



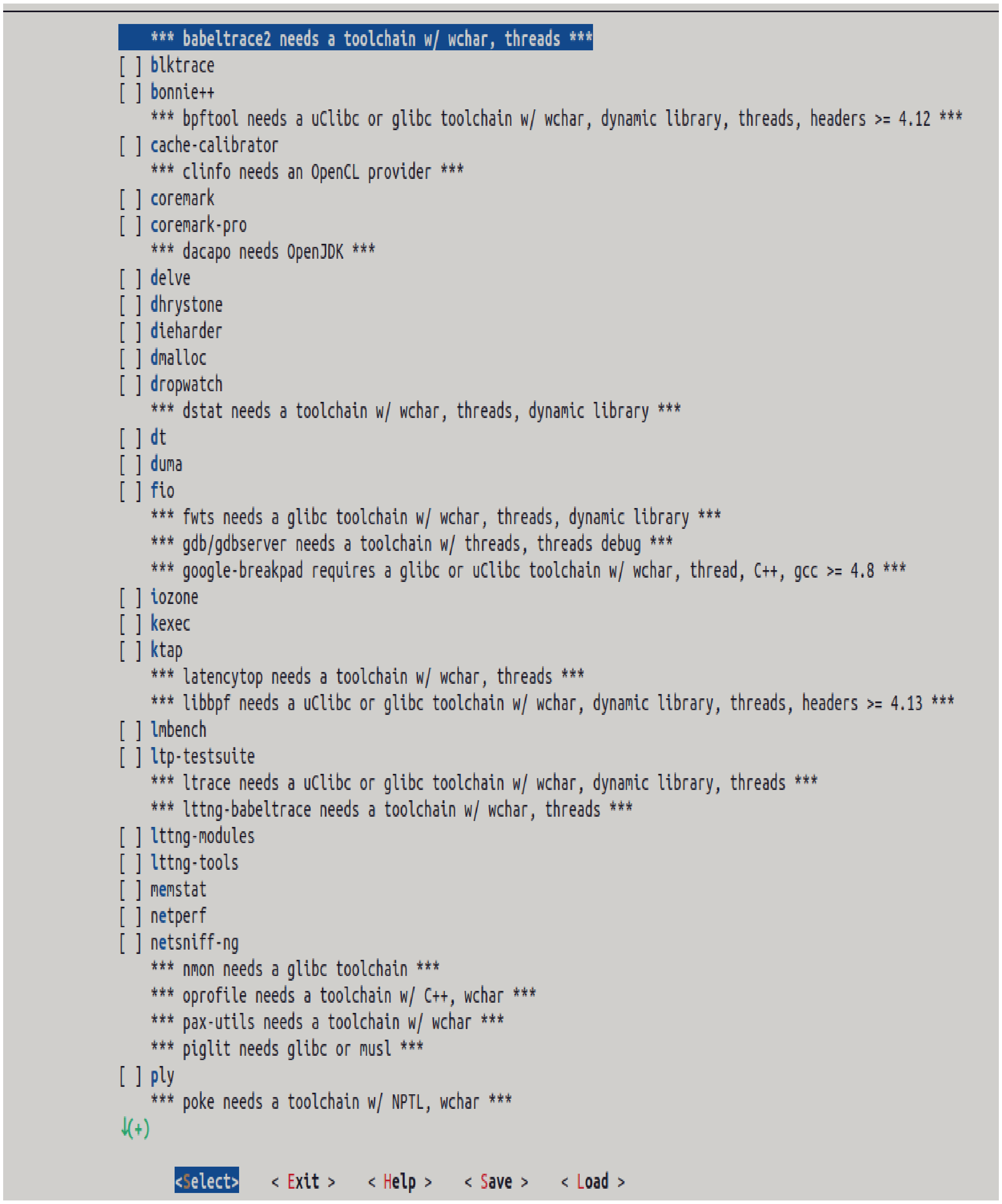
```
    Toolchain type (Buildroot toolchain)  --->
    *** Toolchain Buildroot Options ***
(buildroot) custom toolchain vendor name
    C library (uClibc-ng)  --->
    *** Kernel Header Options ***
    Kernel Headers (Same as kernel being built)  --->
    Custom kernel headers series (5.10.x)  --->
    *** uClibc Options ***
(package/uclibc/uClibc-ng.config) uClibc configuration file to use?
()  Additional uClibc configuration fragment files
[ ] Enable WCHAR support
[ ] Enable toolchain locale/i18n support
    Thread library implementation (Native POSIX Threading (NPTL))  --->
[ ] Thread library debugging
[ ] Enable stack protection support
[*] Compile and install uClibc utilities
    *** Binutils Options ***
    Binutils Version (binutils 2.36.1)  --->
()  Additional binutils options
    *** GCC Options ***
    GCC compiler Version (gcc 10.x)  --->
()  Additional gcc options
[*] Enable C++ support
[ ] Enable Fortran support
[ ] Enable compiler link-time-optimization support
[ ] Enable compiler OpenMP support
[ ] Enable graphite support
    *** Host GDB Options ***
[ ] Build cross gdb for the host
    *** Toolchain Generic Options ***
()  Extra toolchain libraries to be copied to target
[*] Enable MMU support
()  Target Optimizations
()  Target linker options
[ ] Register toolchain within Eclipse Buildroot plug-in
```



```
    Target options  --->
    Build options  --->
    Toolchain  --->
    System configuration  --->
    Kernel  --->
    Target packages  --->
    Filesystem images  --->
    Bootloaders  --->
    Host utilities  --->
    Legacy config options  --->
```



```
-*- BusyBox
(package/busybox/busybox.config) BusyBox configuration file to use?
()    Additional BusyBox configuration fragment files
[ ]   Show packages that are also provided by busybox
[ ]   Individual binaries
[ ]   Install the watchdog daemon startup script
    Audio and video applications  --->
    Compressors and decompressors  --->
    Debugging, profiling and benchmark  --->
    Development tools  --->
    Filesystem and flash utilities  --->
    Fonts, cursors, icons, sounds and themes  --->
    Games  --->
    Graphic libraries and applications (graphic/text)  --->
    Hardware handling  --->
    Interpreter languages and scripting  --->
    Libraries  --->
    Mail  --->
    Miscellaneous  --->
    Networking applications  --->
    Package managers  --->
    Real-Time  --->
    Security  --->
    Shell and utilities  --->
    System tools  --->
    Text editors and viewers  --->
```


```
    *** babeltrace2 needs a toolchain w/ wchar, threads ***
[ ] blktrace
[ ] bonnie++
    *** bpftool needs a uClibc or glibc toolchain w/ wchar, dynamic library, threads, headers >= 4.12 ***
[ ] cache-calibrator
    *** clinfo needs an OpenCL provider ***
[ ] coremark
[ ] coremark-pro
    *** dacapo needs OpenJDK ***
[ ] delve
[ ] dhrystone
[ ] dieharder
[ ] dmalloc
[ ] dropwatch
    *** dstat needs a toolchain w/ wchar, threads, dynamic library ***
[ ] dt
[ ] duma
[ ] fio
    *** fwts needs a glibc toolchain w/ wchar, threads, dynamic library ***
    *** gdb/gdbserver needs a toolchain w/ threads, threads debug ***
    *** google-breakpad requires a glibc or uClibc toolchain w/ wchar, thread, C++, gcc >= 4.8 ***
[ ] iozone
[ ] kexec
[ ] ktap
    *** latencytop needs a toolchain w/ wchar, threads ***
    *** libbpf needs a uClibc or glibc toolchain w/ wchar, dynamic library, threads, headers >= 4.13 ***
[ ] lmbench
[ ] ltp-testsuite
    *** ltrace needs a uClibc or glibc toolchain w/ wchar, dynamic library, threads ***
    *** lttng-babeltrace needs a toolchain w/ wchar, threads ***
[ ] lttng-modules
[ ] lttng-tools
[ ] memstat
[ ] netperf
[ ] netsniff-ng
    *** nmon needs a glibc toolchain ***
    *** oprofile needs a toolchain w/ C++, wchar ***
    *** pax-utils needs a toolchain w/ wchar ***
    *** piglit needs glibc or musl ***
[ ] ply
    *** poke needs a toolchain w/ NPTL, wchar ***
↓(+)

      <Select>    < Exit >    < Help >    < Save >    < Load >
```

## What is BusyBox in Linux?

BusyBox is a software suite that provides several Unix utilities in a single executable file. It is commonly used in embedded systems and in systems with limited resources, such as routers, IoT devices, and small Linux distributions

BusyBox combines several commonly used Unix utilities, such as shell, grep, ls, cp, and many others, into a single binary file that can be used as a drop-in replacement for the original utilities.

BusyBox includes utilities such as ls, cp, mv, grep, and many others, and provides a lightweight alternative to larger and more complex packages such as GNU Coreutils. The utilities in BusyBox are designed to be small and efficient, and many of them can perform multiple functions, which helps to keep the size of the executable file small.

BusyBox can be used as the default command shell for a system, or it can be used as a standalone executable for running specific commands. It is often included in embedded Linux systems and can also be used on desktop systems.

Overall, BusyBox provides a convenient way to have a minimal set of Unix utilities in a single executable file, which can be useful in various contexts where size and efficiency are important considerations.

## BUSYBOX ABI’s:

ABI (Application Binary Interface) refers to the interface between a program and the operating system or runtime environment it is running on.

BusyBox supports multiple ABIs, which are determined by the target architecture and compiler used to build the BusyBox binary.

Some of the ABIs supported by BusyBox include:

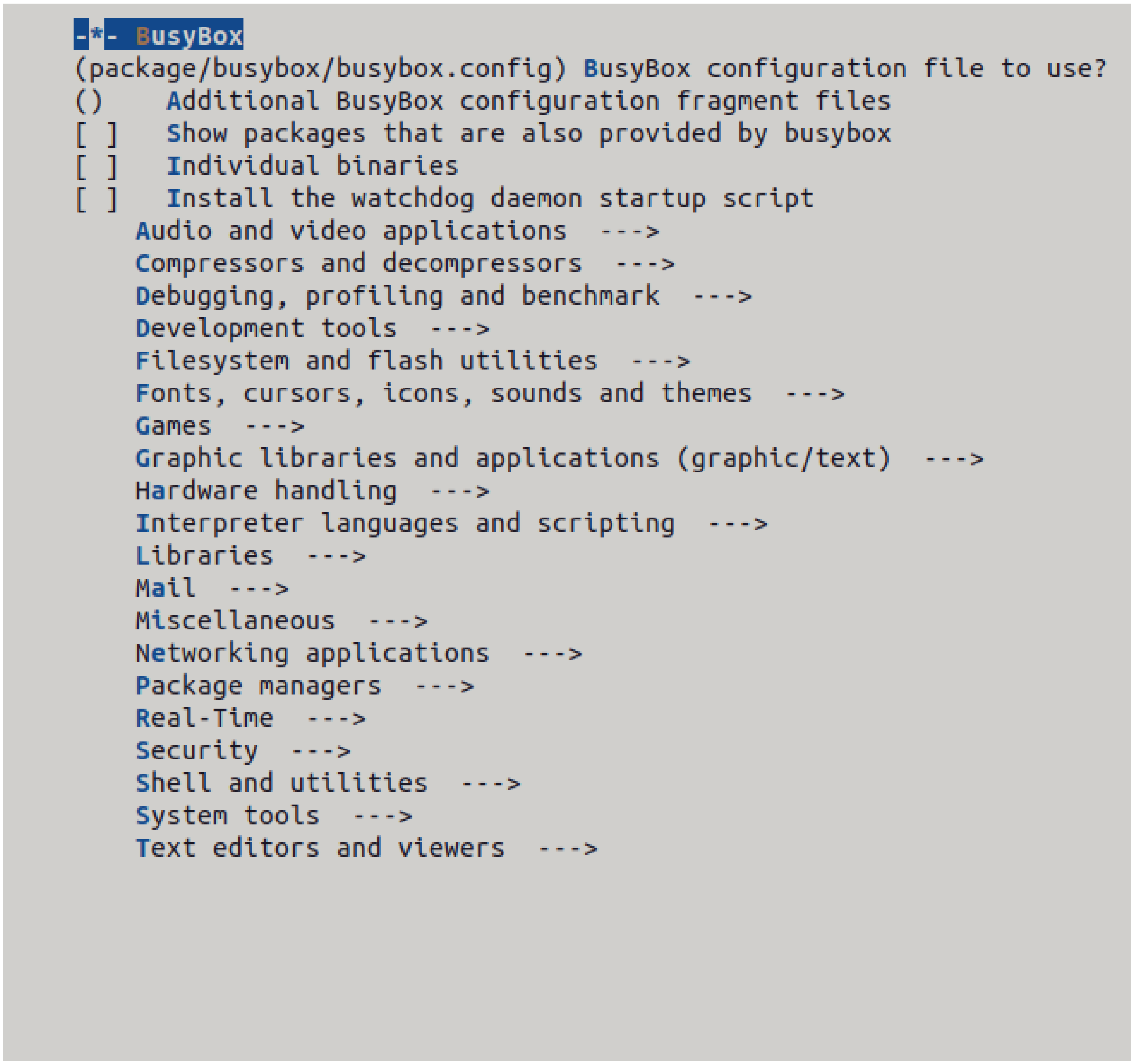
ARM EABI (ARM Embedded Application Binary Interface): This is the default ABI used for ARM-based systems.

x86 (32-bit and 64-bit): BusyBox supports x86 processors, including 32-bit and 64-bit versions of the ABI.

MIPS: BusyBox can be built for MIPS processors, which are commonly used in embedded systems.

PowerPC: BusyBox supports PowerPC processors, which are often used in embedded systems, gaming consoles, and   other specialized devices.

SPARC: BusyBox can be built for SPARC processors, which are often used in servers and other high-performance computing systems.



```
-*- BusyBox
(package/busybox/busybox.config) BusyBox configuration file to use?
()    Additional BusyBox configuration fragment files
[ ]   Show packages that are also provided by busybox
[ ]   Individual binaries
[ ]   Install the watchdog daemon startup script
    Audio and video applications  --->
    Compressors and decompressors  --->
    Debugging, profiling and benchmark  --->
    Development tools  --->
    Filesystem and flash utilities  --->
    Fonts, cursors, icons, sounds and themes  --->
    Games  --->
    Graphic libraries and applications (graphic/text)  --->
    Hardware handling  --->
    Interpreter languages and scripting  --->
    Libraries  --->
    Mail  --->
    Miscellaneous  --->
    Networking applications  --->
    Package managers  --->
    Real-Time  --->
    Security  --->
    Shell and utilities  --->
    System tools  --->
    Text editors and viewers  --->
```

## Advantages of Remote Debugging:

**Remote debugging:** GDBserver allows remote debugging of a program running on a target system. This is particularly useful when the target system is a different architecture than the development system, or when the target system is a separate embedded device that cannot be directly connected to the development system.

**Flexibility:** GDBserver is a flexible tool that can be used with a wide variety of programming languages and development environments. It can be used to debug C and C++ programs, as well as programs written in other languages that can be compiled to run on the target system.

**Real-time debugging:** GDBserver allows for real-time debugging of programs running on a target system. This means that developers can debug programs while they are running, making it easier to identify and fix bugs in complex or time-critical applications.

**Improved productivity:** Using GDBserver can improve productivity by reducing the time required to identify and fix bugs in software. It can also reduce the need for developers to physically access target systems, which can be time-consuming and costly.

### DIsadvantages:

**Security Risks**: GDBserver creates a network connection between the target system and the GDB client, which can potentially create security risks. If the connection is not properly secured, it could be vulnerable to unauthorized access or exploitation.

**Performance Overhead:** The use of GDBserver can introduce some performance overhead, particularly when debugging remote systems. The additional network latency and processing overhead required for remote debugging can affect the performance of the target system.

**Debugging Limitations**: GDBserver provides a limited set of debugging features compared to running GDB locally on the target system. This is because some features, such as debugging kernel code, are not supported by GDBserver.

**Compatibility Issues:** GDBserver may not be compatible with all target systems or architectures, which can limit its usefulness in certain situations.

## Applications:

- ★ Embedded system development.
- ★ Cross-platform development.
- ★ Real-time debugging.
- ★ Kernel debugging.
- ★ Remote debugging.

Reference:

https://sourceware.org/gdb/onlinedocs/gdb/Server.html

https://ftp.gnu.org/old-gnu/Manuals/gdb/html_node/gdb_130.html

https://www.thegeekstuff.com/2014/04/gdbserver-example/