# Android Debug Bridge

Android Debug Bridge (adb) is a versatile command line tool that lets you communicate with an emulator instance or connected Android-powered device. It is a client-server program that includes three components:

- A client, which runs on your development machine. You can invoke a client from a shell by issuing an adb command. Other Android tools such as the ADT plugin and DDMS also create adb clients.
- A server, which runs as a background process on your development machine. The server manages communication between the client and the adb daemon running on an emulator or device.
- A daemon, which runs as a background process on each emulator or device instance.

You can find the adb tool in `<sdk>/platform-tools/`.

When you start an adb client, the client first checks whether there is an adb server process already running. If there isn't, it starts the server process. When the server starts, it binds to local TCP port 5037 and listens for commands sent from adb clients—all adb clients use port 5037 to communicate with the adb server.

The server then sets up connections to all running emulator/device instances. It locates emulator/device instances by scanning odd-numbered ports in the range 5555 to 5585, the range used by emulators/devices. Where the server finds an adb daemon, it sets up a connection to that port. Note that each emulator/device instance acquires a pair of sequential ports — an even-numbered port for console connections and an odd-numbered port for adb connections. For example:

Emulator 1, console: 5554
Emulator 1, adb: 5555
Emulator 2, console: 5556
Emulator 2, adb: 5557
and so on...

As shown, the emulator instance connected to adb on port 5555 is the same as the instance whose console listens on port 5554.

Once the server has set up connections to all emulator instances, you can use adb commands to access those instances. Because the server manages connections to emulator/device instances and handles commands from multiple adb clients, you can control any emulator/device instance from any client (or from a script).

**IN THIS DOCUMENT**



## Enabling adb Debugging

In order to use adb with a device connected over USB, you must enable **USB debugging** in the device system settings, under **Developer options**.

On Android 4.2 and higher, the Developer options screen is hidden by default. To make it visible, go to **Settings > About phone** and tap **Build number** seven times. Return to the previous screen to find **Developer options** at the bottom.

On some devices, the Developer options screen may be located or named differently.

> **Note:** When you connect a device running Android 4.2.2 or higher to your computer, the system shows a dialog asking whether to accept an RSA key that allows debugging through this computer. This security mechanism protects user devices because it ensures that USB debugging and other adb commands cannot be executed unless you're able to unlock the device and acknowledge the dialog. This requires that you have adb version 1.0.31 (available with SDK Platform-tools r16.0.1 and higher) in order to debug on a device running Android 4.2.2 or higher.

For more information about connecting to a device over USB, read Using Hardware Devices (/tools/device.html).

## Syntax

You can issue adb commands from a command line on your development machine or from a script. The usage is:

```
adb [-d|-e|-s <serialNumber>] <command>
```

If there's only one emulator running or only one device connected, the adb command is sent to that device by default. If multiple emulators are running and/or multiple devices are attached, you need to use the `-d`, `-e`, or `-s` option to specify the target device to which the command should be directed.

## Commands

The table below lists all of the supported adb commands and explains their meaning and usage.

Table 1. Available adb commands

| Category | Command | Description | Comments |
|---|---|---|---|
| Target Device | `-d` | Direct an adb command to the only attached USB device. | Returns an error if more than one USB device is attached. |
| | `-e` | Direct an adb command to the only running emulator instance. | Returns an error if more than one emulator instance is running. |
| | `-s <serialNumber>` | Direct an adb command a specific emulator/device instance, referred to by its adb-assigned serial number (such as "emulator-5556"). | See Directing Commands to a Specific Emulator/Device Instance. |
| | `devices` | Prints a list of all attached emulator/device instances. | See Querying for Emulator/Device Instances for more information. |
| General | `help` | Prints a list of supported adb commands. | |
| | `version` | Prints the adb version number. | |
| | `logcat [option] [filter-specs]` | Prints log data to the screen. | |
| | `bugreport` | Prints `dumpsys`, `dumpstate`, and `logcat` data to the screen, for the purposes of bug reporting. | |
| Debug | `jdwp` | Prints a list of available JDWP processes on a given device. | You can use the `forward jdwp:<pid>` port-forwarding specification to connect to a specific JDWP process. For example:<br>`adb forward tcp:8000 jdwp:472`<br>`jdb -attach localhost:8000` |
| Data | `install <path-to-apk>` | Pushes an Android application (specified as a full path to an .apk file) to an emulator/device. | |
| | `pull <remote> <local>` | Copies a specified file from an emulator/device instance to your development computer. | |
| | `push <local> <remote>` | Copies a specified file from your development computer to an emulator/device instance. | |
| | `forward <local> <remote>` | Forwards socket connections from a specified local port to a specified remote port on the emulator/device instance. | Port specifications can use these schemes:<br>• `tcp:<portnum>`<br>• `local:<UNIX domain socket name>`<br>`dev:<character device name>`<br>• `jdwp:<pid>` |
| | | Run PPP over USB.<br>• `<tty>` — the tty for PPP | |

| | | | |
|---|---|---|---|
| Ports and Networking | `ppp <tty> [parm]...` | stream. For example `dev:/dev/omap_csmi_ttyl`.<br>• `[parm]...` — zero or more PPP/PPPD options, such as `defaultroute`, `local`, `notty`, etc.<br>Note that you should not automatically start a PPP connection. | |
| Scripting | `get-serialno` | Prints the adb instance serial number string. | See Querying for Emulator/Device Instances for more information. |
| | `get-state` | Prints the adb state of an emulator/device instance. | You can prepend this command to other adb commands, in which case adb will wait until the emulator/device instance is connected before issuing the other commands. Here's an example:<br>`adb wait-for-device shell getprop` |
| | `wait-for-device` | Blocks execution until the device is online — that is, until the instance state is `device`. | Note that this command does *not* cause adb to wait until the entire system is fully booted. For that reason, you should not prepend it to other commands that require a fully booted system. As an example, the `install` requires the Android package manager, which is available only after the system is fully booted. A command such as<br>`adb wait-for-device install <app>.apk`<br>would issue the `install` command as soon as the emulator or device instance connected to the adb server, but before the Android system was fully booted, so it would result in an error. |
| Server | `start-server` | Checks whether the adb server process is running and starts it, if not. | |
| | `kill-server` | Terminates the adb server process. | |
| Shell | `shell` | Starts a remote shell in the target emulator/device instance. | See Issuing Shell Commands for more information. |
| | `shell [shellCommand]` | Issues a shell command in the target emulator/device instance and then exits the remote shell. | |

## Querying for Emulator/Device Instances

Before issuing adb commands, it is helpful to know what emulator/device instances are connected to the adb server. You can generate a list of attached emulators/devices using the `devices` command:

```
adb devices
```

In response, adb prints this status information for each instance:

- Serial number — A string created by adb to uniquely identify an emulator/device instance by its console port number. The format of the serial number is `<type>-<consolePort>`. Here's an example serial number: `emulator-5554`
- State — The connection state of the instance may be one of the following:
  - `offline` — the instance is not connected to adb or is not responding.
  - `device` — the instance is now connected to the adb server. Note that this state does not imply that the Android system is

fully booted and operational, since the instance connects to adb while the system is still booting. However, after boot-up, this is the normal operational state of an emulator/device instance.
- `no device` — there is no emulator/device connected.

The output for each instance is formatted like this:

```
[serialNumber] [state]
```

Here's an example showing the `devices` command and its output:

```
adb devices
List of devices attached
emulator-5554  device
emulator-5556  device
emulator-5558  device
```

## Directing Commands to a Specific Emulator/Device Instance

If multiple emulator/device instances are running, you must specify a target instance when issuing adb commands. To do so, use the `-s` option in the commands. The usage for the `-s` option is:

```
adb -s <serialNumber> <command>
```

As shown, you specify the target instance for a command using its adb-assigned serial number. You can use the `devices` command to obtain the serial numbers of running emulator/device instances. For example:

```
adb -s emulator-5556 install helloWorld.apk
```

Note that, if you issue a command without specifying a target emulator/device instance while multiple devices are available, adb generates an error.

If you have multiple devices available (hardware or emulated), but only one is an emulator, simply use the `-e` option to send commands to the emulator. Likewise if there's multiple devices but only one hardware device attached, use the `-d` option to send commands to the hardware device.

## Installing an Application

You can use adb to copy an application from your development computer and install it on an emulator/device instance. To do so, use the `install` command. With the command, you must specify the path to the .apk file that you want to install:

```
adb install <path_to_apk>
```

For more information about how to create an .apk file that you can install on an emulator/device instance, see Building and Running (/tools/building/index.html)

Note that, if you are using the Eclipse IDE and have the ADT plugin installed, you do not need to use adb (or aapt) directly to install your application on the emulator/device. Instead, the ADT plugin handles the packaging and installation of the application for you.

## Forwarding Ports

You can use the `forward` command to set up arbitrary port forwarding — forwarding of requests on a specific host port to a different port on an emulator/device instance. Here's how you would set up forwarding of host port 6100 to emulator/device port 7100:

```
adb forward tcp:6100 tcp:7100
```

You can also use adb to set up forwarding to named abstract UNIX domain sockets, as illustrated here:

```
adb forward tcp:6100 local:logd
```

## Copying Files to or from an Emulator/Device Instance

You can use the adb commands `pull` and `push` to copy files to and from an emulator/device instance. Unlike the `install` command, which only copies an APK file to a specific location, the `pull` and `push` commands let you copy arbitrary directories and files to any location in an emulator/device instance.

To copy a file or directory (and its sub-directories) *from* the emulator or device, use

```
adb pull <remote> <local>
```

To copy a file or directory (and its sub-directories) *to* the emulator or device, use

```
adb push <local> <remote>
```

In the commands, `<local>` and `<remote>` refer to the paths to the target files/directory on your development machine (local) and on the emulator/device instance (remote). For example:

```
adb push foo.txt /sdcard/foo.txt
```

## Issuing Shell Commands

Adb provides a Unix shell that you can use to run a variety of commands on an emulator or connected device. The command binaries are stored in the file system of the emulator or device, at `/system/bin/...`

Two of the most common command tools are activity manager (#am) (`am`) and package manager (#pm) (`pm`).

You can use the `shell` command to issue commands, with or without entering the adb remote shell on the emulator/device. To issue a single command without entering a remote shell, use the `shell` command like this:

```
adb [-d|-e|-s <serialNumber>] shell <shell_command>
```

Or enter a remote shell on an emulator/device like this:

```
adb [-d|-e|-s <serialNumber>] shell
```

When you are ready to exit the remote shell, press CTRL+D or type `exit`.

### Using activity manager (am)

Within an adb shell, you can issue commands with the activity manager (`am`) tool to perform various system actions, such as start an activity, force-stop a process, broadcast an intent, modify the device screen properties, and more. While in a shell, the syntax is:

```
am <command>
```

You can also issue an activity manager command directly from adb without entering a remote shell. For example:

```
adb shell am start -a android.intent.action.VIEW
```

Table 2. Available activity manager commands

| Command | Description |
| --- | --- |
| | Start an `Activity` specified by `<INTENT>`. |

| | |
|---|---|
| | See the Specification for <INTENT> arguments (#IntentSpec).<br><br>Options are:<br>• -D: Enable debugging.<br>• -W: Wait for launch to complete.<br>• --start-profiler <FILE>: Start profiler and send results to <FILE>.<br>• -P <FILE>: Like --start-profiler, but profiling stops when the app goes idle.<br>• -R: Repeat the activity launch <COUNT> times. Prior to each repeat, the top activity will be finished.<br>• -S: Force stop the target app before starting the activity.<br>• --opengl-trace: Enable tracing of OpenGL functions.<br>• --user <USER_ID> \| current: Specify which user to run as; if not specified, then run as the current user. |
| start [options] <INTENT> | |
| | Start the Service specified by <INTENT>.<br><br>See the Specification for <INTENT> arguments (#IntentSpec).<br><br>Options are:<br>• --user <USER_ID> \| current: Specify which user to run as; if not specified, then run as the current user. |
| startservice [options] <INTENT> | |
| | Force stop everything associated with <PACKAGE> (the app's package name). |
| force-stop <PACKAGE> | |
| | Kill all processes associated with <PACKAGE> (the app's package name). This command kills only processes that are safe to kill and that will not impact the user experience.<br><br>Options are:<br>• --user <USER_ID> \| all \| current: Specify user whose processes to kill; all users if not specified. |
| kill [options] <PACKAGE> | |
| | Kill all background processes. |
| kill-all | |
| | Issue a broadcast intent.<br><br>See the Specification for <INTENT> arguments (#IntentSpec).<br><br>Options are:<br>• [--user <USER_ID> \| all \| current]: Specify which user to send to; if not specified then send to all users. |
| broadcast [options] <INTENT> | |
| | Start monitoring with an Instrumentation instance. Typically the target <COMPONENT> is the form <TEST_PACKAGE>/<RUNNER_CLASS>.<br><br>Options are:<br>• -r: Print raw results (otherwise decode <REPORT_KEY_STREAMRESULT>). Use with [-e perf true] to generate raw output for performance measurements.<br>• -e <NAME> <VALUE>: Set argument <NAME> to <VALUE>. For test runners a common form is -e <testrunner_flag> <value>[,<value>...].<br>• -p <FILE>: Write profiling data to <FILE>.<br>• -w: Wait for instrumentation to finish before returning. Required for test runners.<br>• --no-window-animation: Turn off window animations while running.<br>• --user <USER_ID> \| current: Specify which user instrumentation runs in; current user if not specified. |
| instrument [options] <COMPONENT> | |
| profile start <PROCESS> <FILE> | Start profiler on <PROCESS>, write results to <FILE>. |
| profile stop <PROCESS> | Stop profiler on <PROCESS>. |
| | Dump the heap of <PROCESS>, write to <FILE>.<br><br>Options are:<br>• --user [<USER_ID>\|current]: When supplying a process name, specify user of process to dump; uses current user if not specified.<br>• -n: Dump native heap instead of managed heap. |
| dumpheap [options] <PROCESS> <FILE> | |
| | Set application <PACKAGE> to debug. |

| | |
|---|---|
| `set-debug-app [options] <PACKAGE>` | Options are:<br>• `-w`: Wait for debugger when application starts.<br>• `--persistent`: Retain this value. |
| `clear-debug-app` | Clear the package previous set for debugging with `set-debug-app`. |
| `monitor [options]` | Start monitoring for crashes or ANRs.<br>Options are:<br>• `--gdb`: Start gdbserv on the given port at crash/ANR. |
| `screen-compat [on\|off] <PACKAGE>` | Control screen compatibility mode of `<PACKAGE>`. |
| `display-size [reset\|<WxH>]` | Override emulator/device display size. This command is helpful for testing your app across different screen sizes by mimicking a small screen resolution using a device with a large screen, and vice versa.<br>Example:<br>`am display-size 1280x800` |
| `display-density <dpi>` | Override emulator/device display density. This command is helpful for testing your app across different screen densities on high-density screen environment using a low density screen, and vice versa.<br>Example:<br>`am display-density 480` |
| `to-uri <INTENT>` | Print the given intent specification as a URI.<br>See the Specification for <INTENT> arguments (#IntentSpec). |
| `to-intent-uri <INTENT>` | Print the given intent specification as an `intent:` URI.<br>See the Specification for <INTENT> arguments (#IntentSpec). |

**Specification for <INTENT> arguments**

### Using package manager (pm)

Within an adb shell, you can issue commands with the package manager (`pm`) tool to perform actions and queries on application packages installed on the device. While in a shell, the syntax is:

```
pm <command>
```

You can also issue a package manager command directly from adb without entering a remote shell. For example:

```
adb shell pm uninstall com.example.MyApp
```

Table 3. Available package manager commands.

| Command | Description |
|---|---|
| `list packages [options] <FILTER>` | Prints all packages, optionally only those whose package name contains the text in `<FILTER>`.<br>Options:<br>• `-f`: See their associated file.<br>• `-d`: Filter to only show disabled packages.<br>• `-e`: Filter to only show enabled packages.<br>• `-s`: Filter to only show system packages.<br>• `-3`: Filter to only show third party packages.<br>• `-i`: See the installer for the packages.<br>• `-u`: Also include uninstalled packages.<br>• `--user <USER_ID>`: The user space to query. |
| `list permission-groups` | Prints all known permission groups. |
| | Prints all known permissions, optionally only those in `<GROUP>`. |

| | |
|---|---|
| `list permissions [options] <GROUP>` | Options:<br>• `-g`: Organize by group.<br>• `-f`: Print all information.<br>• `-s`: Short summary.<br>• `-d`: Only list dangerous permissions.<br>• `-u`: List only the permissions users will see. |
| `list instrumentation` | List all test packages.<br>Options:<br>• `-f`: List the APK file for the test package.<br>• `<TARGET_PACKAGE>`: List test packages for only this app. |
| `list features` | Prints all features of the system. |
| `list libraries` | Prints all the libraries supported by the current device. |
| `list users` | Prints all users on the system. |
| `path <PACKAGE>` | Print the path to the APK of the given `<PACKAGE>`. |
| `install [options] <PATH>` | Installs a package (specified by `<PATH>`) to the system.<br>Options:<br>• `-l`: Install the package with forward lock.<br>• `-r`: Reinstall an exisiting app, keeping its data.<br>• `-t`: Allow test APKs to be installed.<br>• `-i <INSTALLER_PACKAGE_NAME>`: Specify the installer package name.<br>• `-s`: Install package on the shared mass storage (such as sdcard).<br>• `-f`: Install package on the internal system memory.<br>• `-d`: Allow version code downgrade. |
| `uninstall [options] <PACKAGE>` | Removes a package from the system.<br>Options:<br>• `-k`: Keep the data and cache directories around after package removal. |
| `clear <PACKAGE>` | Deletes all data associated with a package. |
| `enable <PACKAGE_OR_COMPONENT>` | Enable the given package or component (written as "package/class"). |
| `disable <PACKAGE_OR_COMPONENT>` | Disable the given package or component (written as "package/class"). |
| `disable-user [options] <PACKAGE_OR_COMPONENT>` | Options:<br>• `--user <USER_ID>`: The user to disable. |
| `grant <PACKAGE_PERMISSION>` | Grant permissions to applications. Only optional permissions the application has declared can be granted. |
| `revoke <PACKAGE_PERMISSION>` | Revoke permissions to applications. Only optional permissions the application has declared can be revoked. |
| `set-install-location <LOCATION>` | Changes the default install location. Location values:<br>• `0`: Auto—Let system decide the best location.<br>• `1`: Internal—install on internal device storage.<br>• `2`: External—install on external media.<br>> **Note**: This is only intended for debugging; using this can cause applications to break and other undesireable behavior. |
| `get-install-location` | Returns the current install location. Return values:<br>• `0 [auto]`: Lets system decide the best location<br>• `1 [internal]`: Installs on internal device storage<br>• `2 [external]`: Installs on external media |
| `set-permission-enforced <PERMISSION> [true\|false]` | Specifies whether the given permission should be enforced. |
| `trim-caches <DESIRED_FREE_SPACE>` | Trim cache files to reach the given free space. |
| | Create a new user with the given `<USER_NAME>`, printing the new |

| | |
|---|---|
| `create-user <USER_NAME>` | user identifier of the user. |
| `remove-user <USER_ID>` | Remove the user with the given `<USER_IDENTIFIER>`, deleting all data associated with that user |
| `get-max-users` | Prints the maximum number of users supported by the device. |

### Examining sqlite3 databases from a remote shell

From an adb remote shell, you can use the sqlite3 (http://www.sqlite.org/sqlite.html) command-line program to manage SQLite databases created by Android applications. The `sqlite3` tool includes many useful commands, such as `.dump` to print out the contents of a table and `.schema` to print the SQL CREATE statement for an existing table. The tool also gives you the ability to execute SQLite commands on the fly.

To use `sqlite3`, enter a remote shell on the emulator instance, as described above, then invoke the tool using the `sqlite3` command. Optionally, when invoking `sqlite3` you can specify the full path to the database you want to explore. Emulator/device instances store SQLite3 databases in the folder `/data/data/<package_name>/databases/`.

Here's an example:

```
adb -s emulator-5554 shell
# sqlite3 /data/data/com.example.google.rss.rssexample/databases/rssitems.db
SQLite version 3.3.12
Enter ".help" for instructions
.... enter commands, then quit...
sqlite> .exit
```

Once you've invoked `sqlite3`, you can issue `sqlite3` commands in the shell. To exit and return to the adb remote shell, use `exit` or `CTRL+D`.

### Recording a device screen

The `screenrecord` command is a shell utility for recording the display of devices running Android 4.4 (API level 19) and higher. The utility records screen activity to an MPEG-4 file, which you can then download and use as part of a video presentation. This utility is useful for developers who want to create promotional or training videos without using a separate recording device.

To use the `screenrecord` from the command line, type the following:

```
$ adb shell screenrecord /sdcard/demo.mp4
```

Stop the screen recording by pressing Ctrl-C, otherwise the recording stops automatically at three minutes or the time limit set by `--time-limit`.

Here's an example recording session, using the adb shell to record the video and the `pull` command to download the file from the device:

```
$ adb shell
shell@ $ screenrecord --verbose /sdcard/demo.mp4
(press Ctrl-C to stop)
shell@ $ exit
$ adb pull /sdcard/demo.mp4
```

The `screenrecord` utility can record at any supported resolution and bit rate you request, while retaining the aspect ratio of the device display. The utility records at the native display resolution and orientation by default, with a maximum length of three minutes.

There are some known limitations of the `screenrecord` utility that you should be aware of when using it:

- Some devices may not be able to record at their native display resolution. If you encounter problems with screen recording, try using a lower screen resolution.
- Rotation of the screen during recording is not supported. If the screen does rotate during recording, some of the screen is cut off in the recording.
- Audio is not recorded with the video file.

Table 4. `screenrecord` options

| Options | Description |
|---|---|

| | |
|---|---|
| `--help` | Displays a usage summary. |
| `--size <WIDTHxHEIGHT>` | Sets the video size, for example: `1280x720`. The default value is the device's main display resolution (if supported), 1280x720 if not. For best results, use a size supported by your device's Advanced Video Coding (AVC) encoder. |
| `--bit-rate <RATE>` | Sets the video bit rate for the video, in megabits per second. The default value is 4Mbps. You can increase the bit rate to improve video quality or lower it for smaller movie files. The following example sets the recording bit rate to 6Mbps:<br>`screenrecord --bit-rate 6000000 /sdcard/demo.mp4` |
| `--time-limit <TIME>` | Sets the maximum recording time, in seconds. The default and maximum value is 180 (3 minutes). |
| `--rotate` | Rotates the output 90 degrees. This feature is experimental. |
| `--verbose` | Displays log information on command line screen. If you do not set this option, the utility does not display any information while running. |

### UI/Application Exerciser Monkey

The Monkey is a program that runs on your emulator or device and generates pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events. You can use the Monkey to stress-test applications that you are developing, in a random yet repeatable manner.

The simplest way to use the monkey is with the following command, which launches your application and sends 500 pseudo-random events to it.

```
adb shell monkey -v -p your.package.name 500
```

For more information about command options for Monkey, see the complete UI/Application Exerciser Monkey (/tools/help/monkey.html) documentation page.

### Other shell commands

For a list of all the available shell programs, use the following command:

```
adb shell ls /system/bin
```

Help is available for most of the commands.

Table 5 lists some of the more common adb shell commands.

Table 5. Some other adb shell commands

| Shell Command | Description | Comments |
|---|---|---|
| `dumpsys` | Dumps system data to the screen. | |
| `dumpstate` | Dumps state to a file. | |
| `logcat [option]... [filter-spec]...` | Enables system and app logging and prints output to the screen. | The Dalvik Debug Monitor Server (DDMS) tool offers integrated debug environment that you may find easier to use. |
| `dmesg` | Prints kernel debugging messages to the screen. | |
| `start` | Starts (restarts) an emulator/device instance. | |
| `stop` | Stops execution of an emulator/device instance. | |

## Enabling logcat logging

The Android logging system provides a mechanism for collecting and viewing system debug output. Logs from various applications and portions of the system are collected in a series of circular buffers, which then can be viewed and filtered by the `logcat` command.

You can use the `logcat` command to view and follow the contents of the system's log buffers. The general usage is:

```
[adb] logcat [option] ... [filter-spec] ...
```

You can use the `logcat` command from your development computer or from a remote adb shell in an emulator/device instance. To view log output in your development computer, you use

```
adb logcat
```

and from a remote adb shell you use

```
logcat
```

See Reading and Writing Logs (/tools/debugging/debugging-log.html) for complete information about logcat commend options and filter specifications.

## Stopping the adb server

In some cases, you might need to terminate the adb server process and then restart it. For example, if adb does not respond to a command, you can terminate the server and restart it and that may resolve the problem.

To stop the adb server, use the `kill-server` command. You can then restart the server by issuing any other adb command.

## Wireless usage

adb is usually used over USB. However, it is also possible to use over Wi-Fi, as described here.

1. Connect Android device and adb host computer to a common Wi-Fi network accessible to both. We have found that not all access points are suitable; you may need to use an access point whose firewall is configured properly to support adb.
2. Connect the device with USB cable to host.
3. Make sure adb is running in USB mode on host.

   ```
   $ adb usb
   restarting in USB mode
   ```

4. Connect to the device over USB.

   ```
   $ adb devices
   List of devices attached
   ######## device
   ```

5. Restart host adb in tcpip mode.

   ```
   $ adb tcpip 5555
   restarting in TCP mode port: 5555
   ```

6. Find out the IP address of the Android device: Settings -> About tablet -> Status -> IP address. Remember the IP address, of the form `#.#.#.#`.
7. Connect adb host to device:

   ```
   $ adb connect #.#.#.#
   connected to #.#.#.#:5555
   ```

8. Remove USB cable from device, and confirm you can still access device:

   ```
   $ adb devices
   List of devices attached
   #.#.#.#:5555 device
   ```

You're now good to go!

If the adb connection is ever lost:

1. Make sure that your host is still connected to the same Wi-Fi network your Android device is.
2. Reconnect by executing the "adb connect" step again.
3. Or if that doesn't work, reset your adb host:

   ```
   adb kill-server
   ```

   and then start over from the beginning.