1. What is unit testing, functional testing, module testing, integration testing?
2. We have CRC, why we go for CMAC?
3. What is checksum?

A checksum is a count of the number of bits in a transmission unit that is included with the unit so that the receiver can check to see whether the same number of bits arrived.

1. What is autosar, why is it useful?
2. Difference between checksum and CRC?
3. What is pre compile, post build, link time ?

* **Pre-compile configuration** − Configuration parameters can not be changed after compilation − Example: Mapping of microcontroller pins to signals
* **Link-time configuration** − Configuration is determined by linker scripts − Configuration parameters can not be changed after link process − Purpose: provides capability to deliver object code to the integrator
* **Post-build configuration** –

1. **Post-build time loadable**

--Configuration parameters can be changed after build process without complete re-flash of ECU

1. **Post-build time selectable**

--Configuration parameter set is selected from multiple configuration sets during boot time

-- All possible configuration sets need to be included at compile time

-- Configuration parameters are stored at a known memory location

--Post-build configuration class BSW modules might also contain pre-compile or link-time parameters (not all parameters have to be post-build)

--Purpose: use one software package in different vehicle

1. What is the difference between compiling and linking?

**Compilation** refers to the processing of source code files (.c, .cc, or .cpp) and the creation of an 'object' file.  if you compile (but don't link) three separate files, you will have three object files created as output, each with the name <filename>.o or <filename>.obj (the extension will depend on your compiler). Each of these files contains a translation of your source code file into a machine language file -- but you can't run them yet! You need to turn them into executables your operating system can use. That's where the linker comes in.

**Linking** refers to the creation of a single executable file from multiple object files. In this step, it is common that the linker will complain about undefined functions (commonly, main itself). During compilation, if the compiler could not find the definition for a particular function, it would just assume that the function was defined in another file. If this isn't the case, there's no way the compiler would know -- it doesn't look at the contents of more than one file at a time. The linker, on the other hand, may look at multiple files and try to find references for the functions that weren't mentioned.

|  |  |  |
| --- | --- | --- |
| **Git task** | **Notes** | **Git commands** |
| [**Tell Git who you are**](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-config) | Configure the author name and email address to be used with your commits.  Note that Git [strips some characters](http://stackoverflow.com/questions/26159274/is-it-possible-to-have-a-trailing-period-in-user-name-in-git/26219423#26219423) (for example trailing periods) from user.name. | git config --global user.name "Sam Smith"  git config --global user.email sam@example.com |
| [**Create a new local repository**](http://atlassian.com/git/tutorial/git-basics#!init) |  | git init |
| [**Check out a repository**](http://atlassian.com/git/tutorial/git-basics#!clone) | Create a working copy of a local repository: | git clone /path/to/repository |
| For a remote server, use: | git clone username@host:/path/to/repository |
| [**Add files**](http://atlassian.com/git/tutorial/git-basics#!add) | Add one or more files to staging (index): | git add <filename>  git add \* |
| [**Commit**](http://atlassian.com/git/tutorial/git-basics#!commit) | Commit changes to head (but not yet to the remote repository): | git commit -m "Commit message" |
| Commit any files you've added with git add, and also commit any files you've changed since then: | git commit -a |
| [**Push**](http://atlassian.com/git/tutorial/remote-repositories#!push) | Send changes to the master branch of your remote repository: | git push origin master |
| [**Status**](http://atlassian.com/git/tutorial/git-basics#!status) | List the files you've changed and those you still need to add or commit: | git status |
| [**Connect to a remote repository**](http://atlassian.com/git/tutorial/remote-repositories#!remote) | If you haven't connected your local repository to a remote server, add the server to be able to push to it: | git remote add origin <server> |
| List all currently configured remote repositories: | git remote -v |
| [**Branches**](http://atlassian.com/git/tutorial/git-branches) | Create a new branch and switch to it: | git checkout -b <branchname> |
| Switch from one branch to another: | git checkout <branchname> |
| List all the branches in your repo, and also tell you what branch you're currently in: | git branch |
| Delete the feature branch: | git branch -d <branchname> |
| Push the branch to your remote repository, so others can use it: | git push origin <branchname> |
| Push all branches to your remote repository: | git push --all origin |
| Delete a branch on your remote repository: | git push origin :<branchname> |
| [**Update from the remote repository**](http://atlassian.com/git/tutorial/remote-repositories) | Fetch and merge changes on the remote server to your working directory: | git pull |
| To merge a different branch into your active branch: | git merge <branchname> |
| View all the merge conflicts:  View the conflicts against the base file:  Preview changes, before merging: | git diff  git diff --base <filename>  git diff <sourcebranch> <targetbranch> |
| After you have manually resolved any conflicts, you mark the changed file: | git add <filename> |
| **Tags** | You can use tagging to mark a significant changeset, such as a release: | git tag 1.0.0 <commitID> |
| CommitId is the leading characters of the changeset ID, up to 10, but must be unique. Get the ID using: | git log |
| Push all tags to remote repository: | git push --tags origin |
| [**Undo local changes**](http://atlassian.com/git/tutorial/undoing-changes) | If you mess up, you can replace the changes in your working tree with the last content in head:  Changes already added to the index, as well as new files, will be kept. | git checkout -- <filename> |
| Instead, to drop all your local changes and commits, fetch the latest history from the server and point your local master branch at it, do this: | git fetch origin  git reset --hard origin/master |
| **Search** | Search the working directory for foo(): | git grep "foo()" |