Compiling, building, and testing SixTrack

K. Sjobak, J. Molson

Jan. 2017

The SixTrack code supports a wide variety of compile time options, compilers, and build environments. Further, many of these options require various libraries to be present, some of which must be built by the user. This document presents what is supported, and how to build the most common versions. Furthermore, an important sub-task of building SixTrack is to test that the built binary is correct; this is also described.

Contents

1 Downloading SixTrack

The last stable release of SixTrack should be found on GitHub, in the SixTrack repository under the SixTrack organization: `https://github.com/SixTrack/SixTrack`. This provides the source code, which you will then have to compile as described in Section 3.

In order to download a release, you can either clone the git repository from our GitHub page, or download a tarball. Since tarballs cannot be automatically updated, and there is no way to check if something has been modified after unpacking, **it is strongly recommended to use git**. If you still want to download a tarball, the newest version is found at
`https://github.com/SixTrack/SixTrack/archive/master.zip`, and older versions can be found through `https://github.com/SixTrack/SixTrack/releases`.

Assuming that you have a command line `git` client installed, you can clone the repository anonymously using the command:

```
1 git clone https://github.com/SixTrack/SixTrack.git
```

You shall now get a new folder "SixTrack" within your current working directory. This folder contains the files from the newest version of SixTrack's "master" branch. Furthermore, it also contains a full clone of the original repository, meaning that you can check out old versions, create branches, commit changes, etc. completely off line.

Assuming that no files tracked by version control have been modified, updating your local copy is done by running the command

```
1 git pull
```

anywhere in the repository folder. If you want to modify SixTrack, get feedback on your changes, and possibly contribute your changes back to the master branch; please also see Section 2.

2 Contributing to SixTrack

The SixTrack code is being used and developed by a wide community, which makes keeping the code coherent between different development branches a challenge. In the past, the main development took place in an SVN repository, to which the main developers could commit what would become the new release versions. Alongside these "official" releases, it was common practice that various new features were developed in separate branches, sometimes in separate repositories, but often in a home-directory of the user/developer writing and running the code. This lead to that these features were never integrated into the main release, resulting in a significant duplication of effort, bugs fixed in one version remaining unfixed in other versions for a long time, new features being unavailable to users of other branches or being lost, and fewer eyes on the code implementing new features.

A switch from SVN to GIT was therefore investigated and implemented, including the preservation of the old commit history. This development tool has a greatly improved support for working with branches compared to SVN, meaning that each new feature, bugfix etc. can be worked on in a separate branch. These branches are very easy to create, and allows the contributor to use a version control tool throughout a potentially complex development. When the development in a branch is finished, integration back to the master branch can be accomplished with a merge, something which is usually a trivial operation. This way, we avoid much of the feature leakage present in the old process.

GIT is a distributed version control system, meaning that each user has and manipulates full copy of the history in a local directory; one can commit, create branches, merge etc. completely off-line. However, a central main repository is still useful for synchronizing progress from different developers and distributing the code (as described in Section 1). CERN IT provides this based on GitLab [1], however this service can only be accessed by internal CERN users. For projects that have external contributors, they therefore recommend to use GitHub [2]. Hosting the project with GitHub has many advantages, as they provide a very good web interface to the repository and an issue tracker including "pull requests". Furthermore,

GitHub has millions of users world wide, which means that solutions to common problems are easy to find.

For the rest of this section, some familiarity with the use of GIT and GitHub is expected of the reader; many guides for this can be found on-line. Still, a brief description of how to configure your GIT client is given in Section 2.1. Note that if you are using a very old GIT client, such as the one installed on LxPlus, some of the commands given in this section may differ.

For SixTrack, we have chosen the following development paradigm:

1. SixTrack development, including support packages such as SixDesk, is organized under a GitHub organization at `https://github.com/SixTrack/`. These organizations are similar to service accounts.

2. The SixTrack main repository is found at `https://github.com/SixTrack/SixTrack`. From here, users can browse the sources, see commit logs, view/edit the issue tracker etc. If a user wants to use the standard version of the code, is possible to clone[1] this repository directly as described in Section 2.

3. Any development should take place in a branch in a personal fork. In order to create such a fork, the user must be logged in to a personal GitHub account. Creating the fork is then a matter of clicking the "fork" button on the web-page for the main repository. The user can then quickly clone this fork to the local computer.

4. In the local clone of the user's fork, the user can then create a new branch from the master branch and start working, committing the changes as the work progresses, and pushing the commits back to their personal fork hosted at GitHub.

5. After finishing the code changes, passing the tests (See Section 5), adding new tests, and writing the documentation, the developer submits a "pull request" for the branch to be merged into the main release branch. This triggers a code review by other developers, possibly recommending some further changes, and which usually leads to the branch being merged. It is sometimes useful to submit the pull request early, as it makes tracking changes from master, getting feedback from the core developers etc. very easy.

6. If it is a long time since the branch diverged from master, the contributor will usually be asked to merge the current master branch into their branch, resolve any conflicts, and re-test the code.

7. Furthermore, if the changes since the last version bump is considered significant by the development group, the version number should be incremented. The developer should also add her/his name to the list of authors at the top of the `sixtrack.s` file. Note that increasing the version number, updating the date, and adding the name should be done just before merging, at the request of the core developers, after merging the current master into the branch. This is to avoid merge conflicts when handling multiple branches.

8. Once the branch is merged, anyone who clones or pulls from the main repository will then get the new feature.

This process makes the development transparent and interactive, promoting collaboration between developers, and makes sure that all code that goes in the main release is reviewed, tested, and documented. The review process is also advantageous for the feature developer, as it makes it easy to get advice on how to implement the new feature and integrate it with existing functionality, and it reduces the chance of software bugs which can make the physics results of the study for which the feature was developed invalid.

When working on a branch in a local clone of a GitHub fork, it is important to keep it up to date with the main upstream repository. This can be done by adding the main upstream repository as as a `remote`,

[1] Cloning a repository creates a new repository that is full copy of the old one, including history. This replaces the "checkout" process of SVN.

in addition to the personal fork which is automatically added as the remote `origin`. When this has been done, one can `fetch` (download) the changes from the main upstream repository, merge these into the master branch of the local repository, and then `push` (upload and update a remote branch) the changes to the private fork, bringing it up to date with the main upstream repository. Using the command-line GIT client, this is done as follows:

1. First time only: Add the `upstream` remote to the local clone:

```
git remote add upstream git@github.com:SixTrack/SixTrack.git
```

2. Download the changes from the upstream:

```
git fetch upstream
```

3. Switch to the `master` branch:

```
git checkout master
```

4. Make sure that you are on the `master` branch:

```
git checkout master
```

   It should say "On branch master", if not then do not proceed.

5. Merge the upstream changes onto the local master:

```
git merge upstream/master
```

   It should either say "Already up to date" or something like "Fast-forwarded"; if not then do not proceed.

6. Update the master branch in your fork with the changes in your local clone:

```
git push
```

   If it asks about which remote to use, use the `origin` remote, which is the name of the remote corresponding to your personal fork.

This process is necessary to do before starting a new branch (which is done by branching the `master` branch in the local clone) , in order to make sure that you start from the newest version. Furthermore, it is often useful to get an updated local master in order to update a long-running branch. Such updates are done by checking out the branch in question, and then running

```
git merge master
```

in order to merge the current status of the `master` branch onto the branch in question. Note that **all development should take place in a new branch**; one should **never** commit directly to master.

2.1 Configuring your GIT client and GitHub account

3 Building SixTrack

The SixTrack source code is located in the "SixTrack" sub-directory of the "SixTrack" repository. SixTrack is normally configured and built using CMake, and for simplicity a wrapper `cmake_six` is provided. This allows configuring the various build options, changing compilers, and changing build types. To run it, simply execute the command:

```
./cmake_six compiler buildtype OPTION1 OPTION2 -OPTION3
```

Here the `compiler` argument specifies the compiler to use, and the `buildtype` argument whether to build a `release` or `debug` version. To take the default compiler and build type, simply leave these options out.

The other options (all UPPER CASE) switch on or off code features as described in Section 3.2. To see the list of options and their meaning, simply run:

```
./cmake_six help
```

Note that to switch an option off put a "-" sign in front of its name, i.e. `-OPTION`.

Each build will produce a new subfolder
`SixTrack_cmakesix_OPTION1_OPTION2_NOOPTION3_compiler_buildtype` ,
and each of the folders will contain a binary named something like
`SixTrack_VERSION_feature1_feature2_cmake_COMPILER_static_32bit|64bit` .
To clear all such folders, simply run:

```
./cmake_six clean
```

3.1 Supported compilers

Most of the code is written in Fortran, where we require Fortran 2003 support. The currently supported compilers are:

- gfortran
- ifort
- nagfor

Furthermore, some of the support libraries are written in C/C++. We here support the following compilers:

- GNU gcc/g++
- Intel icc/icpc
- LLVM clang/clang++

These can be combined, by setting the compiler flags as shown by:

```
./cmake_six help
```

Note that one may also use whatever compilers are the system default by using the `defaultcompiler` flag to `cmake_six`.

Also note that some of these compilers may only be able to build either 32- or 64-bit executables, either due to limitations in the compiler itself, or due to which libraries have been installed on the machine you are compiling on.

3.1.1 gfortran and gcc

The gfortran and gcc compiler is the default on CERN's LxPlus system and on most Linux systems. The system version (version 4.4.7) is old, but can be used to build both 32- and 64-bit executables.

However the version installed and loaded by default on LxPlus is quite old; it is however possible to load a newer version. This can be accomplished by running their setup script as

```
source /afs/cern.ch/sw/lcg/external/gcc/4.9/x86_64-slc6-gcc49-opt/setup.sh
```

where this version is selected to match the Geant4 version that can be loaded as:

```
source /afs/cern.ch/sw/lcg/external/geant4/10.3/x86_64-slc6-gcc49-opt/CMake-setup.sh
```

Unfortunately, this version only have the libraries for building the 64-bit version of SixTrack.

Installing gfortran and libraries for 32- and 64-bit, static/non-static builds on Fedora and Ubuntu.

3.1.2 ifort and icc/icpc

Another popular set of compilers are Intel's ifort and icc. To load these from AFS, simply run their setup script as:

```
source /afs/cern.ch/sw/IntelSoftware/linux/17-all-setup.sh
```

if you want to produce a 64-bit executable, or

```
source /afs/cern.ch/sw/IntelSoftware/linux/17-all-setup.sh ia32
```

for a 32-bit executable.

However note that if using icc/icpc, it is not possible to compile static 64-bit executables, since Intel does not provide a static version of the `libcilkrts` library for 64-bit.

If running the Intel compilers on LxPlus, it is recommended to first load a newer version CMAKE. This can be accomplished through the commands:

```
export PATH=/afs/cern.ch/sw/lcg/contrib/CMake/3.7.0/Linux-x86_64/bin:$PATH
export CMAKE_ROOT=/afs/cern.ch/sw/lcg/contrib/CMake/3.7.0/Linux-x86_64
```

3.1.3 nagfor

To load nagfor version 6.0 on LxPlus, simply run

```
source /afs/cern.ch/sw/fortran/nag/usenag.bash 6.0
```

before any compilation commands. Both 32- and 64-bit executables are supported, as well as both static- and dynamic linking.

3.2 Common build types

By selecting different sets of flags at compile time, the SixTrack sources can be compiled to different versions, which have different capabilities. This section lists the most common build types; note that it is often possible to combine these features, e.g. to use collimation together with increased particle numbers. If the features are incompatible, then this is detected by the cmake script, which will exit with an error message.

3.2.1 Standard build

This build type, used for generic tracking and dynamic aperture studies, is selected by using the standard build flags. In effect, just run

```
./cmake_six
```

without any flags, possibly except for explicitly selecting a compiler. Note that the standard build uses CRLIBM, described in Section 3.3.2.

3.2.2 Collimation

The collimation version of SixTrack is used for collimation studies. Here, more than 64 particles can be tracked by splitting it up in "packs"; in total a maximum of `maxn=20 000` particles can be used. Furthermore, the collimators can scatter particles; the scattering angle is determined through a Monte Carlo routine. The collimation-specific features are controlled by a `COLL` block in the `fort.3` input file; please see for more information. ref

In order to compile the collimation version, just run:

```
./cmake_six COLLIMAT -CRLIBM
```

Sub versions - merlinscatter, hdf5, etc.

3.2.3 Checkpoint/restart

The checkpoint/restart feature in SixTrack allows the simulation to continue from where it stopped after an abort, instead of restarting from the beginning. This is a vital feature when running on e.g. LHC@Home, and works by saving a checkpoint file every `numlcp` turns. The feature is included by using the `CR` build option, i.e.:

```
./cmake_six CR
```

3.2.4 BOINC support

The BOINC libraries are used when running on LHC@Home. This binary will look for a file `Sixin.zip` when started (unless restart files are present), which it will unpack in order to get the actual input files `fort.2` etc.

In order to build SixTrack with support for this, you must first build BOINC as described in Section 3.3.5. The libarchive library must also be built, please see Section 3.3.6. Once this is done, it is possible to build SixTrack with BOINC support by running:

```
./cmake_six CR BOINC API LIBARCHIVE
```

Note that for debugging, it is possible to build BOINC without the "real" BOINC libraries; a set of surrogate functions are then inserted for these. To do this, simply drop the API flag. This makes it possible to test and debug the BOINC version even if the real BOINC libraries are not available; however libarchive is always needed.

3.2.5 Increased particle numbers

It is possible to track more than 64 particles simultaneously in SixTrack; the main limitation for this is the "binary data files" fort.90, fort.89, etc., as one such file is written per pair of tracked particles. In order to work around this, the "Single Track File" (STF) functionality was invented, which basically packs the contents of these binary data files into a single file `singletrackfile.dat`. This makes it possible to increase the particle number to 2048, and to track this many particles compile with:

```
./cmake_six STF BIGNPART
```

After this, the limiting factor is the size of the executable's BSS section, which contains the Fortran COMMON blocks, when using the "small" code model (2 GB). In order to break out of this limitation, some large matrices that are only needed for thick tracking can be allocated on demand in heap memory, and this is enabled by the `DATAMODS` option. This then makes it possible to track up to 65 536 particles when the `HUGENPART` option is in use; to compile this run:

```
./cmake_six STF DATAMODS HUGENPART
```

Reference sixtrack meeting slides?

3.2.6 DA (differential algebra)

This version allows computing DA maps of the machine. Please contact Frank Schmidt if you intend to use it.

To compile it, simply use:

```
./cmake_six DA NAGLIB
```

Note that the DA version requires the NAGFOR library (Section 3.3.7), a proprietary library of mathematical functions that is available at CERN via AFS.

3.3 Libraries

SixTrack leverages a few libraries in order to run correctly. These are generally not written in Fortran but in C or C++, and must be compiled before or alongside SixTrack and then linked with the final executable. This subsection describes how this is done, including any pitfalls.

3.3.1 Automatic building of "external" libraries

It is possible to build some of the "one time build" libraries automatically using the shell script `buildLibraries.sh`, located in the `SixTrack/SixTrack` folder. This script should know about various platform dependent workarounds etc. Note that it contains hard-coded paths to `zlib.a`, to be used when building libarchive on Windows.

3.3.2 CRLIBM

The CRLIBM library is a replacement for the standard math library normally provided by the system, cite used to compute trigonometric functions, logarithms, etc. It is written in C, and contained in the crlibm sub-folder.

The point of CRLIBM is primarily to ensure that results are consistent across different platforms, compilers, etc., which may provide different versions of libm giving slightly different results.

In addition to the math library and its fortran interface (`ericc.c`), this folder also contains functions for converting strings to double (strtod) and a FORTRAN interface (round_near), and functions to enable/disable x87 extended precision. The enabling/disabling of x87 extended precision is used to force the Intel x87 FPU to use 64-bit precision when storing numbers in internal registers, avoiding that results sometimes are stored with 80-bit precision and sometimes 64-bit. This is necessary for CRLIBM to work correctly, and for SixTrack to give consistent results across different compilers which may make different decisions for how long to keep results in registers etc. Furthermore, 80-bit precision is re-enabled before using `read` with `round='nearest'`.

Note that due to problems in GCC's x87 optimizer (introduced somewhere between version 4.4.7 and 4.8.3), the SSE instruction set should be used for all floating point calculations.

In order to build SixTrack with crlibm support, add the option `CRLIBM` to the `make_six` command line; however note that this is done by default. It will automatically build crlibm from the crlibm folder and link it.

3.3.3 MerlinScatter

3.3.4 HDF5

3.3.5 BOINC

The BOINC library is used for volunteer computing . It must be built separately from SixTrack, and Cite BOINC then linked into SixTrack. Note that BOINC can be build with `buildLibraries.sh` as described in Section 3.3.1.

As some modifications have been done to the upstream BOINC library, the correct version is linked into the repository as a `git submodule` at `SixTrack/SixTrack/boinc` . In order to load the submodule, cite run from anywhere in the SixTrack repository run the commands:

```
git submodule init
git submodule update
```

Then, you must build the BOINC library. This is accomplished by `cd`-ing to the the `boinc` directory, then run:

```
./_autosetup -f
./configure --disable-client --disable-server --disable-manager --disable-boinczip
make
```

Finally, you need to build the BOINC Fortran API; this is done by entering the `api` subdirectory and running:

```
make boinc_api_fortran.o
```

Once this is done, you may compile SixTrack with BOINC support as described in Section 3.2.4.

This procedure has been tested on Linux (Fedora 23), Windows Server 2012 (CYGWIN32/64 on MSYS2), and Mac OS X.

Note that to build in a folder hosted on AFS, you must edit the Makefiles in the `api` and `lib` subfolders and change the variable `LN` from `/usr/bin/ln` to `cp`. This is neccessary because AFS does not support hard links between different folders, and configure does not check correctly for it.

3.3.6 libarchive

The libarchive libary is used to unpack the `Sixin.zip` file for the BOINC version, and is needed for the option `ZIPF` block in `fort.3`. It must be built separately from SixTrack, and then linked into SixTrack. Note that libarchive can be build with `buildLibraries.sh` as described in Section 3.3.1.

As some bugfixes have been done to the upstream libarchive library, the correct version is linked into the repository as a `git submodule` at `SixTrack/SixTrack/libarchive` . In order to load the submodule, cite run from anywhere in the SixTrack repository:

```
git submodule init
git submodule update
```

You must then build the libarchive library, which is done by CMAKE. To do this, create a new folder `SixTrack/SixTrack/libarchive_build` and enter it. You can then configure it as

```
cmake -DCMAKE_BUILD_TYPE=Release -DENABLE_BZip2=OFF -DENABLE_ZLIB=ON -DENABLE_CAT=OFF -
    DENABLE_CPIO=OFF -DENABLE_EXPAT=OFF -DENABLE_INSTALL=OFF -DENABLE_LIBXML2=OFF -DENABLE_LZMA
    =OFF -DENABLE_NETTLE=OFF -DENABLE_OPENSSL=OFF -DENABLE_TAR=OFF -DENABLE_CNG=OFF -
    DENABLE_ICONV=OFF -DENABLE_TEST=OFF (-DZLIB_LIBRARY=\$ZLIB\_PATH) -G "Unix Makefiles" ../
    libarchive -LH
```

where `$ZLIB_PATH` is pointing to the location of `libz.a`, which is needed on Windows in order to build a static executable. Finally, build the library using `make` .

Note that libarchive needs several libraries installed; it is the up to the user to install these. Especially note that libarchive depends on zlib, for which a static version is not available on LxPlus. It is therefore only possible to build a dynamically linked version of SixTrack on LxPlus, by using the flag `-STATIC` on the `cmake_six` command line.

Also note that if libarchive was compiled before installing zlib, it will not be able to uncompress compressed .zip files, and attempting to do so will result in a SixTrack runtime error

```
CRITICAL ERROR in read_archive(): When extracting archive (reading data), err=-25
CRITICAL ERROR in read_archive(): Unsupported ZIP compression method (deflation)
```

when opening Sixin.zip for BOINC. To solve this, make sure zlib is installed, and configure/compile libarchive again.

3.3.7 NAGLIB

The NAGLIB library is used for the DA version. It is found on AFS, in the folder `/afs/cern.ch/sw/nag/mark` cite Note that in order to link with this library (when also using `DA`) when compiling with gfortran or nagfor, dynamic linking (i.e. the flag `-STATIC` to `cmake_six`) is required. If compiling with ifort, it is possible to link (mostly) statically.

3.4 Building on platforms other than Linux

3.4.1 Building on OS X

Since OSX does not come with a fortran compiler, gfortran must be installed. For the build system, it is currently assumed that additional tools are installed via Homebrew - for details see `http://brew.sh/`. The native clang/LLVM C and C++ compiler can be used for the C and C++ components of sixtrack; if this is not already done, the installer for the Homebrew system will take care of this.

Building has been tested on OSX 10.12.2 (Sierra) and 10.11.6 (El Capitan). In addition to homebrew, you need to install several packages. This can be accomplished via the following sequence of commands:

```
brew install gcc
brew install pkg-config
brew install automake
brew install libtool
brew install cmake
brew install gawk
```

Once this is done, you may build SixTrack in the standard way, by first running `buildLibraries.sh` and then `cmake_six`.

Note that if you are building a static version of SixTrack, please check that the output binary is statically linked (excluding `libSystem`) by running `otool -L` on the resulting output binary. There have been issues with dynamic linking of `libquadmath.dylib` and other items in your `/usr/local` folders. There should exist in `/usr/local/lib/gcc/6/` a symlink from `libquadmath.dylib` to `libquadmath.0.dylib`. Re-naming or deleting this link (`libquadmath.dylib`) should stop the linker from always selecting this library, even if static is enabled.

3.4.2 Building on Windows

It is possible to build SixTrack on Windows using the "MSYS2" UNIX-like environment. From here, one can install the required libraries and compilers, which can then be used for building a normal and staticly linked Windows executable. Building in this manner was tested on Windows Server 2012 R2, 64-bit version, with MSYS2 version `msys2_x86_64_20161025` running in VirtualBox.

To do this, first install MSYS2 from `http://msys2.github.io`; you should choose the 64-bit version (assuming that you have 64-bit Windows). This installs 3 Unix-like environments on your machine – `MSYS2` which is used for managing packages and GIT, `MSYS2 MINGW 64-bit` which is used for compiling 64-bit executables, and `MSYS2 MINGW 32-bit` which is used for compiling 32-bit executables. After installing MSYS2, you need to bring it up to date. To do this, first update the `pacman` package manager using:

```
pacman -Sy pacman
```

When it is complete, close the MSYS2 window. Then, open a new MSYS2 window and update the core packages:

```
pacman -Syu
```

Close the window when asked to do so, and open a new window. Finally, do a general update using:

```
pacman -Su
```

You can now start installing packages:

```
pacman -Syu openssh git make
pacman -Syu mingw-w64-x86_64-toolchain mingw64/mingw-w64-x86_64-cmake
pacman -Syu mingw-w64-i686-toolchain mingw32/mingw-w64-i686-cmake
```

Install all the suggested packages when prompted; these packages should be sufficient to compile the basic version of SixTrack. You can now clone the SixTrack repository (in the MSYS2 shell) in order to get the sources, which should give you a new folder `SixTrack`. Once you have this folder you may compile the code as normally (in the MINGW32 or MINGW64 shell). Note that to build the 32-bit version, you should pass the options "`-64BIT 32BIT`" to `cmake_six`.

In order to build the BOINC and LIBARCHIVE libraries, you must first download their sources. Normally this is done automatically by the `buildLibraries.sh` script (Section 3.3.1); however since git does not work correctly in the CYGWIN shells, you must do this step manually in the MSYS2 shell. Therefore run inside the `SixTrack` folder in the MSYS2 shell:

```
git submodule init
git submodule update
```

You also need to install the libraries – again from the MSYS2 shell:

```
pacman -Syu autoconf
pacman -Syu automake
pacman -Syu mingw32/mingw-w64-i686-headers-git mingw64/mingw-w64-x86_64-headers-git
pacman -Syu mingw64/mingw-w64-x86_64-libtool mingw32/mingw-w64-i686-libtool
```

Once this is done, you may run `buildLibraries.sh` in the CYGWIN64/32 shell to build the libraries, and then build the BOINC version of SixTrack as normal.

3.5 Legacy build environments

In addition to the `cmake_six` build script, there is the `make_six`. These tools are used in a similar fashion, the main difference is that `make_six` will automatically change depending options, i.e. switching on collimation will automatically switch crlibm off. The `make_six` script works by modifying the ASTUCE input files before running ASTUCE and DAFOR to produce the FORTRAN files, and then modifying `Makefile_six.template` to contain the correct paths etc. before compiling it.

Additionally, there is a plain makefile, which uses an option resolver written in BASH.

Note that these build environments are considered to be obsolete and may soon be removed.

3.6 Running cmake directly

ccmake etc, as normally done in cmake_six

4 Organization of the SixTrack sources

4.1 Coding standards

4.2 Tools

Two tools are required to pre-process the source code before it can be compiled – astuce and dafor. These binaries must be compiled first, and are then ran by the build system to to convert the `.s` files into compileable Fortran.

4.2.1 Astuce

4.2.2 DAFOR

4.2.3 The "beauty" linters

5 Testing SixTrack

SixTrack supports the `ctest` automatic test environment for black box testing of the build executables. This runs the SixTrack binary automatically over a range of input files, and compares the produced output with the expected output (known as "canonicals"). In order to run `ctest`, add the `BUILD_TESTING` flag to the `cmake_six` command line arguments. For GNU compilers, one can also add the flag `COVERAGE`, which makes it possible to see which lines of the code have been exercised by the test.

Note that when implementing new features, one should always run the tests before the new code can be accepted into the master.

5.1 Running CTEST

For running the most basic tests, simply execute

```
ctest
```

in the folder created by the build command:

```
./cmake_six BUILD_TESTING
```

This will iterate over the tests, running SixTrack for each of them, and check the output for pass/fail.

To run several tests in parallel, use the flag `-j` *`numjobs`*.

To run specific tests, use the flag `-R` *`name`*, where *name* is a regexp contained in the names of the tests you want to run. It is also possible to exclude certain tests using the `-E name` flag, where again *name* is a regexp for the tests to exclude.

The tests are classified as `fast` ($\lesssim$ 60 secconds), `medium` ($\lesssim$ 30 minutes), or `slow`; it is possible to run a group by using the `-L` flag, e.g. `-L fast` .

Note that when running a binary compiled with the CR flag, the test harness will kill the binary several times during the run, in order to check that the results still come out correct.

The tests will be ran in the `SixTest/`*`testname`* sub-folder created by `cmake_six`. It is therefore possible to find the produced files, together with the input- and reference files in this folder. The standard output of the simulation and output comparisons are by default stored in the `Testing/Temporary` subfolder.

5.1.1 Submitting to CDASH

In order to collect the results of all tests in a organized way, there is a CDash page for SixTrack, which can be found at `http://abp-cdash.web.cern.ch/abp-cdash/index.php?project=SixTrack`. This page includes information about which tests are running correctly or failing, and optionally code coverage.

In order to send the statistics for a build to CDash, add the argument `-D Experimental` to the `ctest` command line.

On the page that is linked to above, you may either click on the name of a build to see the test results, or you may click on the percentage of coverage to see the coverage.

5.2 Adding new tests

5.3 Testing tools

In order to manage the binaries,

5.4 Legacy test environment "SixTest"

The legacy test environment SixTest can also be used to test SixTrack binaries. It uses the same tests and canonicals as the CDASH-based tests.

To use it, go to the SixTest folder in the main SixTrack repository. You then need to set the EXECS environment variable to the full path of the executable(s) you want to test, and the TESTS variable to a list of the names of the tests you want to run:

```
export TESTS="bb bbe51 bbe52 bbe571ib0 bbinbb_ntwin1 bnl crabamp dipedge distance
    dynk_globalvars elensidealthck4d elensidealthck6d elensidealthin4d elensidealthin6d
    elensidealthin6d_DYNK eric exact fma frs frs60 javier javier_bignpart lost lostevery
    lostnotilt lostnumxv notilt prob1 prob3 s316 thick4 thick6dblocks thick6ddynk
    thick6dsingles tilt"
```

Note that not all tests are applicable for all binaries, and some of the tests are somewhat buggy.

The tests can then be ran by executing `run_pro` or `run_kill` (for the CR version) with a single argument, the run number. Once a test has finished, one can check that the results matches with the canonicals by executing `check_10` (to check fort.10, produced by the POST block), `check_90` (to check `fort.90`, produced by the tracking and read by the post-processing), `check_stf` (to check `singletrackfile.dat`,

produced instead of `fort.90` if STF is enabled), and `check_extras` to check anything else that is defined in the `extra_checks.txt` file for that test.

Please note that these scripts are deprecated, and are likely to be removed soon.

5.5 Known failing tests

The following tests are currently known to fail:

bb, bb_ntwin1 when compiling with "Debug": This is caused by an underflow in the initial closed orbit search, which is uncovered by trapping the underflow exception.

References

[1] CERN GitLab `https://gitlab.cern.ch/`

[2] KB0003132: When is it appropriate to use CERN GitLab or external services such as Github? `https://cern.service-now.com/service-portal/article.do?n=KB0003132`