MBT for MVS 3.8j
User's Guide

Document Number ML02-0001-1

November 2026

Mike Großmann

First Edition (November 2026)

This edition applies to Version 3 Release 2 of MBT, the MVS build tool (mbt 3.2.0), and to all subsequent releases and modifications until otherwise indicated in new editions.

Comments on this book may be addressed to the issue tracker of the mvslovers/mbt repository on GitHub.

© Copyright Mike Großmann 2026. All rights reserved.

Contents

Figures . . . . . **v**

About This Book . . . . . **vii**
How This Book Is Organized vii
Related Publications vii

Chapter 1. Introducing MBT **1**
What MBT Does 1
Projects, Modules and Libraries 1
One Version of MBT per Project 2
MBT 2 and MBT 3 2

Chapter 2. Installing MBT **3**
What You Need 3
Installing a Release 3
The Directory /.mbt 4
A Token for GitHub 4
Checking the Installation 4

Chapter 3. A First Project **7**
The Project Directory 7
The Program 7
The Project File 7
Building 8
What the Build Leaves 8
Testing on the Workstation 9
Preparing the Deployment 9
What Comes Next 10

Chapter 4. Building **11**
The Example Project 11
What a Build Does 11
What to Build 12
Compiling 13
The Build Stamp 13
Assembler Sources 14
COBOL Sources 14
Load Modules 15
Shared Code: [internal] 16
A Library: [lib] 16
Writable Data in Reentrant Modules 16
An Editor That Knows the Build 17

Chapter 5. Dependencies **19**
Declaring a Dependency 19
Fetching: mbt deps 20
Version Ranges 20
The Lock File 21
Building Against a Working Copy 22

The Toolchain . . . 22
Host Tools . . . 22

Chapter 6. MVS Systems: Targets . . . 25
What a Target Is . . . 25
The File targets.toml . . . 25
Passwords . . . 26
Choosing a Target . . . 26
Checking a Target . . . 26
Targets in CI . . . 27
Operator Commands . . . 27
Moving from .env . . . 28

Chapter 7. Deploying . . . 29
The Development Library . . . 29
What a Deploy Does . . . 29
When the Library Fills Up . . . 30
Compressing on the Restart . . . 31
When a Deploy Fails . . . 32

Chapter 8. Testing . . . 33
What a Test Is . . . 33
Finding the Tests . . . 33
A Test That Differs . . . 34
On the Workstation: mbt test . . . 34
On MVS: mbt test –mvs . . . 34
Fixtures . . . 35
When the Run Itself Fails . . . 35
Everything: mbt check . . . 36
Interactive Tests: mbt test –tso . . . 36

Chapter 9. Packaging and SMP Installation . . . 39
What mbt package Writes . . . 39
Describing the Package . . . 39
Modules That Stay Out of the Package . . . 40
A Module for a System Library . . . 41
The Installation Jobs . . . 41
Libraries Installed by SMP . . . 43
The Function Identifier . . . 44
Prereleases and Throwaway Identifiers . . . 45
Upgrading an Installation . . . 45
Checking an Installation . . . 47
Changing the Package Alone: mbt dist . . . 47
Submitting the Jobs Again . . . 47

Chapter 10. Releasing . . . 49
Versions . . . 49
Releasing: mbt release . . . 49
Prereleases: mbt prerelease . . . 49
The Workflows . . . 50
After the Release . . . 51

Chapter 11. Extending MBT . . . 53
Where the Lua Lives . . . 53
Hooks, Tasks and Commands . . . 53
What the Code Can Reach . . . 54
Plugins . . . 55

Appendix A. Migrating from MBT 2 57
Converting the Project File 57
Removing the Submodule 58
Commands 58
MVS Systems 59
Makefile Rules 59
CI 59
Checking the Result 59
Appendix B. Glossary 61
Index 63

Figures

1. Checking the workstation . . . 4
2. src/hello.c . . . 7
3. mbt.toml . . . 7
4. Building the project . . . 8
5. test/tstadd.c . . . 9
6. Running the test on the workstation . . . 9
7. What a deployment would do . . . 9
8. mbt.toml of the project sums . . . 11
9. Building the project sums . . . 11
10. app/sums.c . . . 13
11. asm/addup.asm . . . 14
12. Writable data in a reentrant module . . . 16
13. src/b64.c . . . 19
14. mbt.toml of the project b64 . . . 19
15. Fetching the dependencies . . . 20
16. Building with a dependency . . . 20
17. mbt.lock . . . 21
18. A pin outside its range . . . 21
19. Moving the pins . . . 21
20. The targets of a workstation . . . 26
21. Is the system there? . . . 26
22. What runs on the system . . . 27
23. A first deploy . . . 30
24. A test that checks colours . . . 36
25. Running it . . . 37
26. Packaging a release . . . 39
27. The installation package of sums . . . 39
28. The accept job after an installation job that ended with 12 at the **APPLYCHK** . . . 42
29. The SYSMOD of sums 1.0.0-dev . . . 43
30. A prerelease without a throwaway identifier . . . 45
31. An allocation job with a symbol that no DD uses . . . 46
32. The allocation job, submitted again . . . 48
33. The installation job, submitted a second time . . . 48

Tables

1. What MBT does, and where . . . 1
2. The archives of an MBT release . . . 3
3. The keys of a load module . . . 15
4. Version ranges . . . 21
5. Three libraries, three owners . . . 29
6. Return codes of mbt test –mvs . . . 35
7. The steps of the installation job . . . 42
8. FMIDs derived from the version . . . 44
9. Where extensions live . . . 53

10. Hook points 53
11. The functions and values of ctx 54
12. MBT 2 and MBT 3 57
13. make targets and mbt commands 58

About This Book

This book shows how to build programs for MVS 3.8j with MBT (mbt), the MVS build tool: how a project is described, how it is built and tested on the workstation with the CC/370 cross-toolchain, how its load modules reach MVS, and how a release is packaged for installation with SMP. Every command and key is described in full in the companion volume, the *MBT Reference*.

How This Book Is Organized

Chapter 1	“Introducing MBT”.
Chapter 2	“Installing MBT”.
Chapter 3	“A First Project”.
Chapter 4	“Building”.
Chapter 5	“Dependencies”.
Chapter 6	“MVS Systems: Targets”.
Chapter 7	“Deploying”.
Chapter 8	“Testing”.
Chapter 9	“Packaging and SMP Installation”.
Chapter 10	“Releasing”.
Chapter 11	“Extending MBT”.
Appendix A	“Migrating from MBT 2”.
Appendix B	“Glossary”.

Related Publications

ML02-0002	*MBT Reference*: every command, every key of **`mbt.toml`**, the files MBT reads and writes, and its Lua API.
ML01-0001	*CC/370 User’s Guide*: the cross-toolchain MBT drives.

Chapter 1. Introducing MBT

MBT is the build tool for programs that run on MVS 3.8j and are built on a workstation. It reads a short description of a project, the file **`mbt.toml`**, and from it compiles, assembles and links the project with the CC/370 cross-toolchain, runs its tests on the workstation and on MVS, sends its load modules to a system to try them, and packages a release so that it can be installed with SMP. This chapter describes what MBT does, which terms it uses, and how its version is chosen for each project.

What MBT Does

A project for MVS has more steps than compiling: C and assembler sources become object modules, object modules become load modules or a library, other projects' libraries have to be fetched in the right version, tests have to run, and the result has to reach MVS. MBT does each of these with one command:

Table 1. What MBT does, and where

Command	What it does	Touches MVS
`mbt build`	compiles, assembles and links the project with CC/370	no
`mbt deps`	fetches the libraries the project depends on, in the versions it pins	no
`mbt test`	builds the tests for the workstation and runs them there	no
`mbt package`	writes the files of a release, including the SMP installation package	no
`mbt release`	sets the version and tags the release	no
`mbt deploy`	sends the load modules to an MVS system and receives them into a development library	yes
`mbt test --mvs`	runs the tests on MVS, in batch and under TSO	yes

The build runs on the workstation. Everything that turns a source into a load module happens on macOS, Linux or Windows, with the commands of CC/370: **`cc370`**, **`as370`**, **`ld370`** and **`ar370`**. MVS is not needed to build a project. Only deploying and testing on MVS reach a system, and they do so through mvsMF, a REST interface that runs on MVS.

MBT is one program. It is installed once on the workstation and serves every project there (Chapter 2, "Installing MBT" on page 3). A project holds no copy of it, no **`Makefile`** and no build scripts: its directory contains the project file, the sources and, once it has dependencies, a lock file.

Projects, Modules and Libraries

MBT builds *projects*. A project is a directory with a file **`mbt.toml`** in it, usually a Git repository. The project file names the project, its version and its kind, and describes what the project produces:

- **Load modules.** Each is a member of a load library on MVS: a program that is run with **`EXEC PGM=`***`name`*, or a module that another program loads. A project can build several. The project file has a table for each, **`[module.`***`NAME`***`]`**, named like the member.
- **A library.** A static library of object modules, which other projects link their programs with. A project builds at most one, described in **`[lib]`**, and publishes it with its releases together with the headers that declare its functions.
- **Tests.** Small programs that check the project, each a load module of its own. MBT finds them under **`test/`** by itself.

The project's *kind* says what it is: an **`application`**, a **`library`**, or a **`module`** such as a server module that another program loads. Only **`library`** changes what MBT does: **`mbt build`** without arguments builds the library of such a project, and the load modules of any other. **`application`** and **`module`** build the same and record what the project is.

Note: To be confirmed: whether **`module`** will come to mean more than **`application`**.

The MVS systems a project is deployed to and tested on are *targets*. They are not part of the project: they belong to the workstation and are described once, in the file **`~/.mbt/targets.toml`** (Chapter 6, "MVS Systems: Targets" on page 25). So every project on the workstation can use them, and no password is kept in a project directory.

One Version of MBT per Project

A project names the version of MBT it is built with:

```
[toolchain]
mbt = "3.0"
```

`"3.0"` means any release 3.0.*x*, **`"3.0.2"`** exactly that one. When the installed **`mbt`** is another version, it fetches the one the project names into **`~/.mbt/versions/`**, checks it against the checksums published with that release, and runs it instead. So every project builds with the MBT it was written for, however many projects share the workstation, and a project that is not ready for a newer MBT keeps its pin. The environment variable **`MBT_NO_SWITCH=1`** keeps the **`mbt`** that was started.

`[toolchain] cc370` and **`[toolchain] libc370`** name the releases of CC/370 and of its C library LIBC/370 that a release of the project is built with. They do not select a toolchain on the workstation, which builds with the one it has installed; MBT only checks that one against them (Chapter 5, "Dependencies" on page 19).

MBT 2 and MBT 3

This book describes MBT 3. Its predecessor, MBT 2, was not a program of its own but a Git submodule in every project, driven by **`make`**, and it read the project from **`project.toml`**. MBT 3 builds the same object modules and load modules from the same sources and can still read a **`project.toml`**. Appendix A, "Migrating from MBT 2" on page 57 shows how to move a project from MBT 2 to MBT 3.

Chapter 2. Installing MBT

MBT is a single program without dependencies. It is installed once on the workstation, and every project there uses it. Besides MBT, a project needs the CC/370 cross-toolchain with LIBC/370, which MBT runs but does not install. This chapter shows how to install MBT, what else it needs, and how to check the result.

What You Need

- **A workstation** running macOS or Linux on an arm64 or amd64 processor, or Windows on amd64.
- **CC/370 with LIBC/370**, installed and on the **`PATH`**: the commands **`cc370`**, **`as370`**, **`ld370`**, **`ar370`** and **`xmit370`**, and the C library in the directory tree of the toolchain. Their installation is described in the *CC/370 User's Guide*, Chapter 2. MBT uses the **`cc370`** it finds on the **`PATH`**, and LIBC/370 in that toolchain's directory tree. The versions a project names in **`[toolchain]`** do not select another one; they are checked ("Checking the Installation" on page 4).
- **A C compiler for the workstation**, such as **`cc`** or **`gcc`**, if you want to run the tests of a project on the workstation (Chapter 8, "Testing" on page 33).
- **Access to GitHub**, when a project has dependencies or pins a version of MBT: MBT fetches both from GitHub releases.
- **An MVS 3.8j system with mvsMF**, only to deploy and to test on MVS. The build itself needs no MVS.

Installing a Release

MBT is published as a release of the GitHub repository **`mvslovers/mbt`**, with one archive per platform and a list of their SHA-256 checksums:

Table 2. The archives of an MBT release

Archive	Platform
`mbt-`*`version`***`-darwin-arm64.tar.gz`**	macOS on Apple silicon
`mbt-`*`version`***`-darwin-amd64.tar.gz`**	macOS on Intel
`mbt-`*`version`***`-linux-amd64.tar.gz`**	Linux on x86-64
`mbt-`*`version`***`-linux-arm64.tar.gz`**	Linux on ARM
`mbt-`*`version`***`-windows-amd64.zip`**	Windows on x86-64
`mbt-`*`version`***`-sha256.txt`**	the checksums of the archives

Download the archive for your platform and the checksums, check the archive, unpack it, and put **`mbt`** into a directory on your **`PATH`**. With the GitHub command **`gh`**, on a Mac with Apple silicon, for MBT 3.2.0:

```
gh release download v3.2.0 -R mvslovers/mbt \
   -p 'mbt-3.2.0-darwin-arm64.tar.gz' -p 'mbt-3.2.0-sha256.txt'
shasum -a 256 -c --ignore-missing mbt-3.2.0-sha256.txt
tar -xzf mbt-3.2.0-darwin-arm64.tar.gz
install mbt-3.2.0-darwin-arm64/mbt ~/.local/bin/mbt
```

The archives can also be downloaded from the release page in a browser. Then check that the shell finds the program:

```
mbt version
```

You install MBT once and never again for a new project. A project that asks for another version of MBT gets it without your help (“One Version of MBT per Project” on page 2): the installed **mbt** fetches it into **~/.mbt/versions/** and runs it.

The Directory /.mbt

MBT keeps what belongs to the workstation rather than to a project in the directory **.mbt** in your home directory:

targets.toml	the MVS systems you work with (Chapter 6, “MVS Systems: Targets” on page 25).
versions/	the versions of MBT that projects asked for.
init.lua	your own extensions, loaded for every project (Chapter 11, “Extending MBT” on page 53).

The environment variable **MBT_HOME** names another directory instead. What MBT keeps for one project, such as the libraries it depends on, is in the directory **.mbt** of that project, which is not committed (“What the Build Leaves” on page 8).

A Token for GitHub

MBT asks the GitHub API when it fetches dependencies, tools, plugins or another version of itself. Without a token, GitHub allows 60 requests an hour, shared by everything on the machine, and with several projects that limit is soon reached: MBT then reports **GitHub API rate limit reached**. Any token lifts the limit, and it needs no permissions. With the **gh** command:

```
export GITHUB_TOKEN=$(gh auth token)
```

Put the line into the start-up file of your shell.

Checking the Installation

mbt doctor checks what MBT needs. Run in a project directory with **--offline**, it checks the workstation alone: the commands of CC/370, the toolchain’s directory tree with LIBC/370, their versions against what the project asks for, and the project file. Without **--offline** it also logs on to the project’s MVS target, read-only. Figure 1 shows the check in the project of Chapter 3, “A First Project” on page 7.

```
$ mbt doctor --offline
[mbt] Running environment checks (mbt 3 / cc370)...
[mbt] cc370: /home/user/.local/bin/cc370
[mbt] as370: /home/user/.local/bin/as370
[mbt] ld370: /home/user/.local/bin/ld370
[mbt] ar370: /home/user/.local/bin/ar370
[mbt] xmit370: /home/user/.local/bin/xmit370
[mbt] sysroot: /home/user/.local/cc370 (libc370: libc.a, crtm.o; cc370: macros, libcc370rt.a OK)
[mbt] libc370 2.6.3 in /home/user/.local/cc370 (no version declared, nothing to check)
[mbt] cc370 1.5.0 (no release pinned in [toolchain] cc370)
[mbt] mbt.toml valid: hello v0.1.0-dev
[mbt] --offline: MVS not contacted
[mbt] All checks passed
```

Figure 1. Checking the workstation

Each line names what was found and where. These are errors, after which **`mbt doctor`** ends with return code 2:

- a command of CC/370 that is missing;
- a directory tree without the C library (**`libc.a`**) or without the macros and the run-time library of CC/370;
- a LIBC/370 older than **`[toolchain] libc370`** asks for;
- an **`mbt.toml`** that cannot be read;
- without **`--offline`**, an MVS target that cannot be reached or refuses the logon.

A CC/370 older than **`[toolchain] cc370`** is only a warning: the build may still work, and **`mbt doctor`** says so without failing.

Chapter 3. A First Project

This chapter builds a small project from nothing: one C program and one test. It writes the project file, builds the load module, runs the test on the workstation and prepares the deployment to MVS. Each step is described in more detail in the chapters that follow; here the aim is to go through all of them once.

You need MBT and CC/370 installed as described in Chapter 2, “Installing MBT” on page 3. No MVS system is needed for this chapter.

The Project Directory

Create a directory **`hello`** with this layout:

```
hello/
  mbt.toml          the project file
  src/hello.c       the program
  test/tstadd.c     a test
```

MBT expects public headers under **`include/`**, which is on the include path without being named, and tests under **`test/`**, where it finds them by itself. The sources live where the project file lists them; **`src/`** is the custom. Make the directory a Git repository (**`git init`**): MBT puts the commit it built into a build stamp that a program can include (Chapter 4, “Building” on page 11), and **`mbt release`** commits, tags and pushes a release. Outside a repository the stamp names no commit, and nothing else fails.

The Program

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      printf("Hello from MVS\n");
6      return 0;
7  }
```

Figure 2. src/hello.c

Nothing in it is particular to MVS. On MVS, **`printf`** writes to the DD statement **`SYSPRINT`**, and the value **`main`** returns becomes the return code of the job step.

The Project File

Figure 3 is the whole description of the project.

```
1  schema = 3
2
3  [project]
4  name = "hello"
```

```
 5  version = "0.1.0-dev"
 6
 7  [toolchain]
 8  mbt = "3.0"
 9
10  [module.HELLO]
11  sources = ["src/hello.c"]
12  rent = false
13  reus = false
14
15  [tests]
16  rent = false
17  reus = false
```

Figure 3. mbt.toml

`schema = 3`	the version of the file format. MBT 3 reads schema 3.
`[project]`	the project's name and version. A version ending in **`-dev`** is a version being worked on, not yet released. The kind of the project is not given, so it is an **`application`**, whose main product is load modules.
`[toolchain]`	the version of MBT the project is built with: any 3.0.*x*.
`[module.HELLO]`	one load module, the member **`HELLO`**, built from **`src/hello.c`**. **`rent`** and **`reus`** say whether the module is reentrant and serially reusable. MBT has no default for them: every module states both (Chapter 4, "Building" on page 11).
`[tests]`	the same two attributes for every test of the project.

The entry point of the module, the start-up code of the C library and the library itself are not named: a C program gets them without being asked.

Building

In the project directory, run **`mbt build`**:

```
$ mbt build
[cc370] src/hello.c
[ld370] HELLO (entry=@@CRT0)
[mbt] Build complete
```

Figure 4. Building the project

MBT compiled **`src/hello.c`** with **`cc370`** and linked the load module **`HELLO`** with **`ld370`**, with the C library and its start-up routine **`@@CRT0`** as entry point. Only what changed since the last build is built again: a second **`mbt build`** right away compiles and links nothing and only reports **`Build complete`**.

What the Build Leaves

The build writes into the directory **`build/`**:

`hello.o`	the object module of **`src/hello.c`**.
`hello.d`	the headers **`hello.c`** included, so that a change to one of them builds it again.
`HELLO`	the load module, as a member of a load library.
`HELLO.iebcopy`	the same member in the form in which it is sent to MVS.

Besides **build/**, MBT keeps a directory **.mbt/** in the project for what it fetches and stages, and **dist/** for the files of a release (Chapter 9, “Packaging and SMP Installation” on page 39). None of the three belongs into the repository. Add them to **.gitignore**:

```
build/
dist/
.mbt/
```

Testing on the Workstation

test/tstadd.c is a test. MBT treats every C or assembler source under **test/** as one, named after its file in upper case: **TSTADD**.

```
 1  #include <mbtcheck.h>
 2
 3  static int add(int a, int b) { return a + b; }
 4
 5  int main(void)
 6  {
 7      CHECK_EQ(add(2, 3), 5, "2 + 3");
 8      CHECK(add(-1, 1) == 0, "-1 + 1 is 0");
 9      return mbt_test_summary("TSTADD");
10  }
```

Figure 5. test/tstadd.c

<mbtcheck.h> comes with MBT. **CHECK** records whether a condition holds, **CHECK_EQ** whether two numbers are equal, each with a description, and **mbt_test_summary** prints the result and returns 0 when every check passed. That return code is all MBT looks at: 0 is a passed test, anything else a failed one.

mbt test compiles each test with the C compiler of the workstation and runs it there:

```
$ mbt test

  TEST       RESULT       ASSERTIONS
  ---------- ------------ --------------
  TSTADD     ok  rc=0     2 pass / 0 fail

  1 host test(s) | assertions: 2 PASS, 0 FAIL
[mbt] all host tests passed
```

Figure 6. Running the test on the workstation

A test written this way is portable C, so the same source also runs on MVS as a load module, in batch and under TSO, with **mbt test --mvs** (Chapter 8, “Testing” on page 33). On the workstation it runs in seconds, which makes it the check to run after every change.

Preparing the Deployment

mbt deploy sends the load modules to an MVS system and receives them into the project’s development library, **HELLO.DEV.LINKLIB** by default. That needs a target (Chapter 6, “MVS Systems: Targets” on page 25). Without one, **--dry-run** shows what would happen, and needs no system and no password:

```
$ mbt deploy --dry-run
[mbt] WARNING: no ~/.mbt/targets.toml: using the mbt 2 settings (MBT_MVS_*, .env) -- 'mbt target import .env --name NAME' turns them into
a target
[mbt] Deploy target: HELLO.DEV.LINKLIB
[mbt] Modules (1): HELLO
[mbt] Packed 1 module(s) -> hello.deploy.xmit
[mbt] [dry-run] would upload hello.deploy.xmit -> IBMUSER.MBT.XMIT.IN
[mbt] [dry-run] would RECEIVE into IBMUSER.HELLO.MBTDPLY, then IEBCOPY replace into HELLO.DEV.LINKLIB (DISP=SHR; allocated first if
missing)
```

Figure 7. What a deployment would do

The warning says that no target is defined yet, so MBT fell back on the settings of MBT 2. With none of those either, it uses their defaults, which is where **`IBMUSER`** in the data set names comes from; Chapter 6, “MVS Systems: Targets” on page 25 shows how to define a target. MBT packed the load module into a TSO TRANSMIT file, which it would upload to MVS, receive into a staging library, and copy from there into the development library. Chapter 7, “Deploying” on page 29 describes the deployment, and why it goes to a development library of its own.

What Comes Next

- Chapter 4, “Building” on page 11 describes the build: more modules, assembler sources, a library, and the attributes of a module.
- Chapter 5, “Dependencies” on page 19 adds libraries of other projects.
- Chapter 6, “MVS Systems: Targets” on page 25 and Chapter 7, “Deploying” on page 29 bring the project to MVS, Chapter 8, “Testing” on page 33 runs its tests there.
- Chapter 9, “Packaging and SMP Installation” on page 39 and Chapter 10, “Releasing” on page 49 turn a version of the project into a release that can be installed with SMP.

Chapter 4. Building

This chapter describes how MBT builds a project: what it compiles and links, with which options, and how a project describes its load modules, its library and the code they share. It uses a project with a C program, a C source and an assembler routine, and a library made of the latter two.

The Example Project

The project **sums** computes the sum and the mean of some numbers. The sum is an assembler routine, **ADDUP**, the mean a C function that calls it, and a C program prints both:

```
sums/
  mbt.toml
  include/sums.h      declares addup() and mean()
  asm/addup.asm       the sum, in assembler
  src/mean.c          the mean, in C
  app/sums.c          the program SUMS
```

```
 1  schema = 3
 2
 3  [project]
 4  name = "sums"
 5  version = "1.0.0-dev"
 6
 7  [toolchain]
 8  mbt = "3.0"
 9
10  # Code shared by the load modules, linked from a private archive.
11  [internal]
12  sources = ["src/*.c", "asm/*.asm"]
13
14  # The same code, published as a library for other projects.
15  [lib]
16  name = "sums"
17  sources = ["src/*.c", "asm/*.asm"]
18  headers = ["include/sums.h"]
19
20  [module.SUMS]
21  sources = ["app/sums.c"]
22  rent = true
23  reus = true
```

Figure 8. mbt.toml of the project sums

addup and **mean** are used twice: by the program of this project, and by other projects, to which **sums** offers them as a library. The project file therefore names them twice, in **[internal]** for its own load modules and in **[lib]** for others. Both are described below.

What a Build Does

Figure 9 shows the build of everything in this project. **--all** builds the library as well as the load modules, and **-v** shows each command instead of a short line per step.

```
$ mbt build --all -v
cc370 -O1 -std=gnu99 -Wall -Wextra -Werror -I include -I .mbt/include -I .mbt -MMD -MP -c app/sums.c -o build/sums.o
cc370 -O1 -std=gnu99 -Wall -Wextra -Werror -I include -I .mbt/include -I .mbt -MMD -MP -c src/mean.c -o build/mean.o
as370 -o build/addup.o asm/addup.asm
ar370 rc build/sumsint.a build/mean.o build/addup.o
ar370 rc build/sums.a build/mean.o build/addup.o
ld370 -L/home/user/.local/cc370/lib -e @@CRT0 build/sums.o build/sumsint.a -lcc370rt -lc -lc --rent --reus -iebcopy -o build/SUMS
[mbt] Build complete
```

Figure 9. Building the project sums

A build has four kinds of steps:

1. **Compiling and assembling.** Every C source is compiled with **cc370**, every **.asm** or **.s** source assembled with **as370**, each into an object module in **build/**. The object module is named after the source, without its directory: **app/sums.c** becomes **build/sums.o**.
2. **Archiving.** The object modules of **[internal]** become the private archive **build/***project***int.a**, and those of **[lib]** the library **build/***name***.a**, with **ar370**.
3. **Checking.** Before the first link, MBT checks that no module declared reentrant holds writable data (“Writable Data in Reentrant Modules” on page 16).
4. **Linking.** Each load module is linked with **ld370** from its own object modules, the private archive, the libraries of the project’s dependencies (Chapter 5, “Dependencies” on page 19) and the C library, into **build/***NAME* and **build/***NAME***.iebcopy**.

Note: Because the directory is dropped, two sources of the same name, such as **src/util.c** and **lib/util.c**, would make the same object module. MBT refuses such a project before it compiles anything, naming both files: give every source of a project its own name. The same source listed by several modules is no conflict.

Independent steps run at the same time, as many as the workstation has processors; **-j** *n* sets another number. A step runs again when one of its inputs changed or a header it includes did, and also when its command changed, for example because a flag in **mbt.toml** did. A build in which nothing changed does nothing.

What counts as an input depends on the source:

a C source	the source, and every header **cc370** read while compiling it.
an assembler source	the source, every macro and **COPY** member **as370** read while assembling it, nested macros included, and every directory on the macro search path: those of **-I** in **asflags**, then the macros of CC/370, LIBC/370 and MACLIB/370. A directory counts through its date, which moves when a member is added to it, removed or renamed. A new member that hides another one of the same name, from a directory searched earlier, rebuilds the object for that reason. So does saving a file in such a directory, the sources themselves included when the project puts them there, for every assembler source that searches it.
a COBOL or PL/I source	the same as for C and for assembler: what the translator read (copybooks, includes) and, for the assembler it wrote, what **as370** read.

MBT records the members in **build/***name***.d** beside the object, in the form **cc370** writes for headers. An object built before MBT recorded them has no such file and is assembled once more.

What to Build

mbt build	the main product: the library of a project of kind **library**, the load modules of any other.
mbt build --all	the load modules and the library.

`mbt build --tests` also the tests, as load modules for MVS. **`mbt test`** builds them for the workstation by itself (Chapter 8, “Testing” on page 33).

`mbt build` *NAME*… only the named load modules or tests. Case does not matter: **`mbt build sums`** builds **`SUMS`**.

`mbt clean` removes **`build/`** and **`dist/`**. **`mbt distclean`** also removes **`.mbt/`**, with the dependencies, tools and plugins MBT staged there; **`mbt.lock`** stays.

Compiling

Every C source is compiled with the same options, in this order:

1. MBT’s own: **`-O1 -std=gnu99 -Wall -Wextra -Werror`**. The code is optimized at the level CC/370 is validated at, the dialect is C99 with the GNU extensions, and every warning stops the build.
2. **`-I include`**, when the project has a directory **`include/`**, then a **`-I`** for each directory in **`[build] include`**.
3. **`[build] cflags`**.
4. A **`-I`** for the headers of each dependency, for MBT’s own headers (**`.mbt/include`**, with **`<mbtcheck.h>`**) and for **`.mbt`** (with **`<buildstamp.h>`**).

```
[build]
include = ["src/private"]           # besides include/
cflags  = ["-DNDEBUG", "-std=gnu89"]
asflags = ["-I", "maclib"]
```

An option in **`cflags`** comes after MBT’s and wins where the two disagree: **`-std=gnu89`** above compiles C89 again, **`-O0`** turns the optimization off. **`asflags`** are passed to **`as370`**, which by itself finds the macros of CC/370 and LIBC/370; give it the directories of the project’s own macros. Changing a member in one of them assembles again every source that read it (Chapter 4, “Building” on page 11).

The Build Stamp

Before it compiles, MBT writes the header **`.mbt/buildstamp.h`**. A program that includes it can say exactly which build it is, as **`app/sums.c`** does:

```
 1  #include <stdio.h>
 2  #include <buildstamp.h>
 3  #include "sums.h"
 4
 5  int main(void)
 6  {
 7      static const int v[] = { 3, 5, 8, 13, 21 };
 8
 9      printf("%s %s (%s)\n", MBT_PROJECT, MBT_VERSION, MBT_COMMIT);
10      printf("sum %d, mean %d\n", addup(5, v), mean(5, v));
11      return 0;
12  }
```

Figure 10. app/sums.c

`MBT_PROJECT` the project’s name, as a string.

`MBT_VERSION` its version, as a string.

`MBT_COMMIT` the short commit of the checkout, followed by **`-dirty`** when tracked files have uncommitted changes, or **`unknown`** outside a Git repository.

`MBT_COMMIT_DIRTY` 1 for uncommitted changes, else 0.

The header holds no time, and MBT rewrites it only when one of the values changed, so an unchanged commit compiles nothing again.

Assembler Sources

A source ending in **`.asm`** or **`.s`** is assembled with **`as370`**. It is listed like a C source and can be part of a load module, of **`[internal]`** or of **`[lib]`**. Figure 11 is the routine of the example. How assembler code and C call each other is described in the *CC/370 User's Guide*, Chapter 5.

```
 1  *        int addup(int n, const int *v) -- sum of v[0] .. v[n-1]
 2  ADDUP    CSECT
 3           STM   14,12,12(13)       save the caller's registers
 4           LR    12,15              R12 = base register
 5           USING ADDUP,12
 6           LM    2,3,0(1)           R2 = n, R3 = v
 7           SR    15,15              R15 = sum = 0
 8           LTR   2,2                no numbers?
 9           BNP   DONE
10  LOOP     A     15,0(,3)           add v[i]
11           LA    3,4(,3)            next element
12           BCT   2,LOOP
13  DONE     L     14,12(,13)         restore the caller's registers,
14           LM    0,12,20(13)        except R15, the result
15           BR    14
16           END
```

Figure 11. asm/addup.asm

`as370` ends with the highest severity of its messages. As on MVS, an assembly with warnings (return code 4) passes and its object module is used, but MBT always shows it: a **`WARNING`** line with **`as370`**'s messages, and a count at the end of the build. A warning can cost an instruction, as a non-blank column 72 that turns the next statement into a continuation does, so a strict project sets **`max_rc = 0`** in **`[build]`**, and any warning fails the build.

COBOL Sources

A module can be written in COBOL. **`cobc370`**, a COBOL-74 compiler that emits S/370 assembler, turns each **`.cbl`** source into assembler, and **`as370`** and **`ld370`** do the rest, as for an assembler module:

```
[build]
cobflags = ["-I", "copy"]          # where COPY finds its copybooks

[module.PAYROLL]
sources = ["src/payroll.cbl"]
entry   = "PAYROLL"                # the PROGRAM-ID
rent    = false
reus    = false
```

The build shows both steps for each source, and the generated assembler stays in **`build/cobol/`** for a look:

```
[cobc370] src/payroll.cbl
[as370] build/cobol/payroll.asm
[ld370] PAYROLL (entry=PAYROLL)
```

A changed copybook translates the programs that copy it again: **`cobc370`** records them, as **`cc370`** records headers. The module carries the COBOL runtime **`cobc370`** emits into it. It needs no LIBC/370 and no C startup, which is why **`entry`** is required. The exception is a module whose main program is C: when one of its C sources defines **`main()`**, the COBOL is called from C, the C startup stays, and **`entry`** is left out. COBOL calls C back through a stub made with LIBC/370's **`CCALL`** macro (from LIBC/370

2.9.0). The system macros the generated code uses (OPEN, DCB, SPIE, TIME and the rest) come from maclib370 beside **as370**.

cobc370 is not part of CC/370. Put it on **PATH**, name it in **MBT_COBC370**, or declare it in **[tools]** (“Host Tools” on page 22). A rebuilt **cobc370** does not by itself translate the sources again.

Load Modules

Each load module is a table **[module.***NAME***]**, *NAME* being the member name, one to eight characters. A name with **#**, **@** or **$** is written in quotes: **[module."IRX#HELO"]**.

Table 3. The keys of a load module

Key	Meaning
sources	the sources of this module alone. A pattern such as **"src/*.c"** stands for every file it matches. Required.
exclude	sources the patterns of **sources** match but the module does not use.
rent	**true** when the module is reentrant: one copy is shared by every task that runs it. Required.
reus	**true** when the module is serially reusable: a copy can be used again once a task has finished with it. Required.
refr	**true** when the module is refreshable: it never changes itself. Default **false**.
ac	the authorization code: **1** for a program that must run APF-authorized. Default **0**.
entry	the entry point. Default **@@CRT0**, the start-up routine of a C program.
startup	how the C run time comes in, see below.
dep_startup	**true** when a dependency provides the start-up routine **@@START**, as the library of an HTTP server does for its CGI modules; **false** keeps that of LIBC/370.
aliases	further names of the module, such as **["REXX", "RX"]**.
ship	**false** keeps the module out of the installation package, though it is built and deployed. Default **true**. See “Modules That Stay Out of the Package” on page 40.

rent and reus have no default. Whether a module may be shared between tasks is a property of its code, and a wrong guess is not caught by any build: it shows on MVS, as an abend or as two tasks overwriting each other’s data. So MBT makes every module say it. Most C programs that keep data in **static** or global variables are **rent = false** and **reus = false**.

The start-up of a C program comes out of the C library: with the entry point **@@CRT0**, the linkage editor finds the start-up routine in **libc.a** and with it everything that runs before **main**. **startup** changes that:

(omitted)	a C program, as above.
false	no C run time: a module with its own entry point, usually in assembler. Set **entry** to it.
"crtm"	a C module that another C program links to or calls: it uses the C run time of its caller instead of starting its own. See the *LIBC/370 Programmer’s Guide*, “The Start-Up”. It needs **crtm.o** in the sysroot, which newer LIBC/370 releases no longer ship; without it the build stops with an error that says so.

Shared Code: [internal]

Code that several load modules of the project use belongs into **[internal]**. Its sources are built once into a private archive, which every load module and every test of the project is linked with. The linkage editor takes from it only the object modules a program refers to, directly or through another, so a module that does not use a function does not carry it.

```
[internal]
sources = ["src/*.c", "asm/*.asm"]
exclude = ["src/debug.c"]
```

The archive is not published: it belongs to the project.

A Library: [lib]

A project that other projects link with describes its library in **[lib]**:

name	the library's name. The archive is **build/***name***.a**.
sources	the sources of the object modules in it.
headers	the headers that declare what it offers. They are published with it, and a project that depends on it compiles with them.

mbt build builds the library for a project of kind **library**, **mbt build --all** for any project. **mbt package** publishes it with the project's release, as an archive of **lib/** and **include/** that other projects fetch (Chapter 5, "Dependencies" on page 19). A library is an archive of object modules, not a load module: it never goes to MVS as it is, but into the load modules that are linked with it.

Many projects build the same sources twice, as **sums** does: in **[internal]** for their own programs, and in **[lib]** for others.

Writable Data in Reentrant Modules

A module declared **rent = true** is one copy that every task running it shares, and MVS may load it into storage that cannot be written to. It must not keep data it changes in itself. Before it links such a module, MBT looks through its sources for data that would land in the module itself: variables defined outside functions, and **static** variables inside them, that are not **const**. In a **rent = true** module that is an error. A module with **ac = 1** is checked too, and gets a warning, since MVS fetches an authorized module into storage the program cannot change. Modules that are neither, and tests, are not checked. Figure 12 shows what happens when **v** in **app/sums.c** loses its **const**.

```
$ mbt build
[cc370] app/sums.c
[mbt] ERROR: writable data in SUMS (rent = true): app/sums.c:7: static int v[] = 3, 5, 8, 13, 21
[mbt] A rent = true module is one copy shared by every task that LINKs it, so it may hold no writable data. Move it onto the heap or the
stack, make it const, or declare rent = false (cc370#100).
[mbt] ERROR: writable data in a rent = true module
```

Figure 12. Writable data in a reentrant module

Nothing is linked then. Make the data **const**, keep it on the stack or in storage the program obtains, or declare the module **rent = false**. **mbt module-data** runs the check alone; it reports the first three findings of each module, **--all** every one.

An Editor That Knows the Build

`mbt compiledb` writes **`compile_commands.json`**, the list of compile commands that editors with a C language server, such as **`clangd`**, read. The editor then finds the headers of CC/370, LIBC/370, the dependencies and the project as the build does. Run it again when the sources or the options change. The file belongs to the workstation; add it to **`.gitignore`**.

Chapter 5. Dependencies

A project can use the libraries of other projects. MBT fetches them from the projects' GitHub releases, keeps the exact version and checksum of each in the file **`mbt.lock`**, and builds with what it fetched. This chapter shows how a project declares a dependency, how the versions are chosen and pinned, how to build against a working copy of another project, and how the toolchain and host tools fit into the same scheme.

Declaring a Dependency

The project **`b64`** encodes a string in base64 with the library of the project **`crypto370`**:

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  #include <string.h>
 4  #include <base64.h>
 5
 6  int main(void)
 7  {
 8      const char *text = "MVS 3.8j";
 9      unsigned char *enc;
10
11      enc = base64_encode((const unsigned char *)text, strlen(text), NULL);
12      if (enc == NULL)
13          return 8;
14      printf("%s -> %s\n", text, enc);
15      free(enc);
16      return 0;
17  }
```

Figure 13. src/b64.c

The header **`<base64.h>`** and the function **`base64_encode`** come from **`crypto370`**. The project file says so in **`[dependencies]`**:

```
 1  schema = 3
 2
 3  [project]
 4  name = "b64"
 5  version = "0.1.0-dev"
 6
 7  [toolchain]
 8  mbt = "3.0"
 9  libc370 = "2.4.0"
10
11  [dependencies]
12  "mvslovers/crypto370" = "^1"
13
14  [module.B64]
15  sources = ["src/b64.c"]
16  rent = false
17  reus = false
```

Figure 14. mbt.toml of the project b64

The key is the GitHub repository, *owner*`/`*repo*, in quotes because of the slash; the value is the range of versions the project works with (“Version Ranges”).

Fetching: mbt deps

`mbt deps` resolves each range against the releases of the repository, downloads the library of the version it chose, and stages it in the project:

```
$ mbt deps
[mbt] mvslovers/crypto370 ^1 -> 1.0.1
[mbt]   staged -> .mbt/deps/crypto370 (sha f5686240265b)
[mbt] Locked 1 dependency(ies) -> mbt.lock
```

Figure 15. Fetching the dependencies

A library release is an archive *repo*`-`*version*`-lib.tar.gz` with two directories, **`include/`** and **`lib/`**. MBT unpacks it into **`.mbt/deps/`***repo*`/`:

```
.mbt/deps/crypto370/include/base64.h
.mbt/deps/crypto370/include/blowfish.h
.mbt/deps/crypto370/include/sha256.h
.mbt/deps/crypto370/lib/crypto370.a
```

From there the build takes it without being told: every **`.mbt/deps/*/include`** is on the include path, and every archive in **`.mbt/deps/*/lib`** is searched when a module is linked, after the project’s own code, with the C library searched before and after them:

```
$ mbt build -v
[mbt] libc370 2.6.3 in /home/user/.local/cc370 (>= 2.4.0)
cc370 -O1 -std=gnu99 -Wall -Wextra -Werror -I .mbt/deps/crypto370/include -I .mbt/include -I .mbt -MMD -MP -c src/b64.c -o build/b64.o
ld370 -L/home/user/.local/cc370/lib -e @@CRT0 build/b64.o -lcc370rt -lc .mbt/deps/crypto370/lib/crypto370.a -lc --norent --noreus -
iebcopy -o build/B64
[mbt] Build complete
```

Figure 16. Building with a dependency

The linkage editor takes from the archive only the object modules the program uses, here the base64 encoder, not SHA-256 or Blowfish.

`mbt build` does not fetch anything itself. When a dependency is declared but not staged, it stops and says so:

```
[mbt] ERROR: dependencies not staged: mvslovers/crypto370 -- run 'mbt deps'
```

Run **`mbt deps`** after cloning a project, after changing **`[dependencies]`**, and after **`mbt distclean`**.

Version Ranges

Versions in the ecosystem have three numbers, *major*`.`*minor*`.`*patch*, optionally followed by **`-dev`** for a version in development or **`-rc`***n* for a release candidate. They are ordered by the numbers, and for equal numbers **`-dev`** comes before **`-rc`***n*, which comes before the release: **`1.0.2-dev`** < **`1.0.2-rc1`** < **`1.0.2`**.

Table 4. Version ranges

Range	Allows
`^1`	1.0.0 and later, below 2.0.0
`^1.4`	1.4.0 and later, below 2.0.0
`^0.3`	0.3.0 and later, below 0.4.0: below 1.0, each minor may break the last
`>=1.2.0`	1.2.0 and every later version
`>=1.2.0,<1.4.0`	both conditions: 1.2.0 up to, not including, 1.4.0
`=1.3.1`	exactly 1.3.1

Prereleases only when asked for. A version with **`-dev`** or **`-rc`***n* is chosen only for a range that names a prerelease itself, such as **`>=1.0.2-dev`**. **`^1`** above chose 1.0.1 although 1.0.2-dev was already published. A project that needs a change not yet released asks for it explicitly, and only until the release is out.

`^` is the range to use normally: a project that works with 1.4 of a library works with every later 1.*x* as well, if the library keeps to the rule that only a new major version may break its callers.

The Lock File

`mbt deps` writes the version it chose and the SHA-256 checksum of the archive into **`mbt.lock`**:

```
{
  "mvslovers/crypto370": {
    "sha256": "f5686240265baf69773e887bb7abd2c3f8da84d77bd377df38f95f4aa879fb6f",
    "version": "1.0.1"
  }
}
```

Figure 17. mbt.lock

Commit `mbt.lock`. From then on, **`mbt deps`** on any workstation and in CI fetches exactly that version and checks that the archive has exactly that checksum. Two builds of the same commit link the same code, whatever was released in between.

Nothing but `mbt deps --update` moves a pin. When the range in **`mbt.toml`** no longer includes the pinned version, **`mbt deps`** stops with return code 3 rather than choose another one. Here the range was changed to **`>=1.0.2-dev`**:

```
$ mbt deps
[mbt] ERROR: mvslovers/crypto370: locked '1.0.1' no longer fits '>=1.0.2-dev' -- run 'mbt deps --update' to re-resolve
(return code 3)
```

Figure 18. A pin outside its range

`mbt deps --update` resolves every range again, fetches what it chose and rewrites the lock:

```
$ mbt deps --update
[mbt] mvslovers/crypto370 >=1.0.2-dev -> 1.0.2-dev
```

```
[mbt]   staged -> .mbt/deps/crypto370 (sha aa14a4bf654e)
[mbt] Locked 1 dependency(ies) -> mbt.lock
```

Figure 19. Moving the pins

The same holds when an archive no longer has the checksum the lock records, which happens when a prerelease is published again under the same version: **`mbt deps`** refuses it until **`mbt deps --update`** accepts the new one. A prerelease that was fetched once stays buildable on that workstation, because MBT keeps the archives it downloaded, by checksum.

A dependency newly added to **`[dependencies]`** is resolved and added to the lock by a plain **`mbt deps`**: that is a deliberate change, not a drift.

Building Against a Working Copy

When two projects change together, the dependent one can build against the other's working copy instead of a release. Name it in **`.mbt/deps.local.toml`**:

```
[override]
"mvslovers/crypto370" = { path = "../crypto370" }
```

`mbt deps` then stages that dependency from the working copy: the library its last build left in **`build/`** and the headers its **`[lib]`** names. Build the library there first; otherwise **`mbt deps`** reports that it is not built. GitHub and the lock are not consulted for it, and **`mbt.lock`** keeps the release pin, so deleting the file returns to the release. **`.mbt/`** is not committed, so the override never reaches anyone else.

The Toolchain

CC/370 and LIBC/370 are not dependencies in this sense: they are installed on the workstation, and every project there builds with the same ones (Chapter 2, "Installing MBT" on page 3). **`[toolchain]`** records which releases the project expects:

```
[toolchain]
mbt     = "3.0"
cc370   = "1.5.0"
libc370 = "2.4.0"
```

`libc370` the oldest LIBC/370 the project builds with. A build on a workstation with an older one stops; Figure 16 on page 20 shows the check passing. Since MBT 3 needs the C start-up inside **`libc.a`**, the value must be 2.3.0 or later.

`cc370` the CC/370 the project expects. An older one on the workstation draws a warning.

The release workflow builds a release with exactly these two (Chapter 10, "Releasing" on page 49); the build workflow for pull requests builds with the current development state of CC/370 and LIBC/370, so that a change there that breaks the project shows before the next release.

Host Tools

Some projects need a program on the workstation besides the toolchain, for example to build a disk image that the project ships. Such a tool can be fetched from a GitHub release and pinned like a dependency:

```
[tools]
ufsd-utils = { repo = "mvslovers/ufsd-utils", version = "1.0.1" }
```

`mbt deps` downloads the archive of the tool for the workstation's platform and records its checksum in **`mbt.lock`**, together with those of the other platforms, so a lock written on a Mac also checks the download of a Linux build machine. The project's Lua code reaches the tool by name (Chapter 11, "Extending MBT" on page 53). The release asset and the member of the archive that is the program follow a naming rule that **`asset`** and **`bin`** can change; the *MBT Reference* describes both.

Chapter 6. MVS Systems: Targets

Deploying and testing on MVS need a system to go to. MBT calls such a system a *target* and keeps the targets of a workstation in one file, **`~/.mbt/targets.toml`**, apart from every project. A project therefore does not say where it is deployed; the workstation does, and the same project can go to a test system today and to another tomorrow. This chapter describes the file, where passwords come from, how a target is chosen, and how to check that a system can be reached.

What a Target Is

A target is the set of ways to reach one MVS system:

mvsMF	the REST interface MBT deploys and tests through: it uploads files, submits jobs and reads their output. Every target has it.
Hercules	the web interface of the Hercules emulator the system runs on. MBT can send operator commands through it when mvsMF cannot (“Operator Commands” on page 27). Optional.
TN3270	a terminal session, for the interactive TSO tests (Chapter 8, “Testing” on page 33). Optional.
SSH	a shell on the machine that runs Hercules. Optional.

Besides these, a target names the high-level qualifier of the data sets MBT allocates there, and the volume and job classes it uses.

The File targets.toml

Each target is a table **`[target.`***`name`***`]`**, with a table for each way to reach it:

```
[target.lab]
default  = true            # the target used without --target
hlq      = "IBMUSER"       # default: the mvsMF user
volume   = "PUB001"        # where RECEIVE allocates
jobclass = "A"             # default A
msgclass = "H"             # default H

[target.lab.mvsmf]         # required
url      = "http://lab:8080"
user     = "IBMUSER"
password = { keychain = "mbt/lab" }

[target.lab.hercules]
url      = "http://lab:8181"

[target.lab.tn3270]
host     = "lab"
user     = "IBMUSER"
password = { keychain = "mbt/lab" }

[target.lab.ssh]
host     = "lab"
user     = "hercules"
```

`volume` names the volume on which MBT allocates the data sets it receives. Without it, the system chooses. Name one on a system that does not choose a suitable volume by itself, such as MVS/CE.

When MBT allocates a library on the named volume, it checks afterwards that the library really is there, because an APF entry names a data set together with its volume. The file is in **~/.mbt**, or in the directory that **MBT_HOME** names. The *MBT Reference* lists every key.

Passwords

A password can be written into the file, but MBT accepts that only while nobody else can read it (**chmod 600**). Better, the file says where the password comes from:

{ env = "LAB_PASSWORD" }	from an environment variable.
{ keychain = "mbt/lab" }	from the macOS Keychain, or from the Secret Service on Linux, under that service name.
{ cmd = ["pass", "show", "mvs/lab"] }	from the first line a program writes. The program is started directly, without a shell.

Store a password in the keychain once, in an interactive terminal: run from a script, the commands store an empty entry without a word.

```
security add-generic-password -s mbt/lab -a IBMUSER -w     # macOS
secret-tool store --label='mbt lab' service mbt/lab        # Linux
```

An empty entry, an unset variable or a failing command is reported as such, not as a failed logon. MBT never prints a password: **mbt target list** and **mbt doctor** show only where it comes from.

Choosing a Target

mbt deploy and **mbt test --mvs** take **--target** *name*, and the environment variable **MBT_TARGET** does the same. An unknown name is an error that lists the targets there are. The name **env** means the target described by environment variables (“Targets in CI” on page 27). Without a name, MBT takes, in this order:

1. the target described by environment variables, when **MBT_TARGET_MVSMF_URL** is set;
2. the target marked **default = true**;
3. the only target in the file.

A file with several targets and none marked default is an error, which asks for **--target** or a default. Only one target may be the default. Only when there is no **targets.toml**, or it defines no target, does MBT fall back to the settings of MBT 2 (“Moving from .env” on page 28).

Checking a Target

mbt target list shows the targets, the default marked with *****:

```
$ mbt target list
* lab          http://lab:8080       IBMUSER    password: keychain mbt/lab hercules ssh tn3270
```

Figure 20. The targets of a workstation

mbt target ping asks every way of reaching a target whether it answers, without logging on. An HTTP 401 from mvsMF is the right answer here: the service is there and wants a logon.

```
$ mbt target ping lab
[mbt] lab        mvsmf     http://lab:8080       answering (HTTP 401, 76 ms)
[mbt] lab        hercules  http://lab:8181       answering (HTTP 200, 2 ms)
[mbt] lab        tn3270    lab:3270              open (1 ms)
[mbt] lab        ssh       lab:22                open (0 ms)
```

Figure 21. Is the system there?

mbt target info logs on to mvsMF and reports what runs there:

```
$ mbt target info lab
[mbt] lab        mvsmf     http://lab:8080       logged on as IBMUSER (HTTP 200) -- 1.1.1-dev lab, MVS 3.8j
[mbt] lab        hercules  http://lab:8181       answering without a logon (HTTP 200) -- Hercules 4.10.0.11773-SDL-DEV-g4675e7e1, built
Sep 5 2026; 3270 terminals: 0 of 33 connected
[mbt] lab        tn3270    lab:3270              open (1 ms)
[mbt] lab        ssh       lab:22                open (1 ms)
```

Figure 22. What runs on the system

--wait *seconds* lets **ping** and **info** wait for a system that is still starting, which is useful in CI. **mbt doctor** without **--offline** logs on to the target too (“Checking the Installation” on page 4).

MBT logs on to mvsMF once per run, at the first request that needs it, so a dry run never logs on. It uses that session for every request and logs off at the end, also after an error or Ctrl-C.

Targets in CI

A CI job has no **~/.mbt**. It describes its target in environment variables instead, the password from a secret of the repository:

MBT_TARGET_MVSMF_URL	the URL of mvsMF. Setting it defines the target.
MBT_TARGET_MVSMF_USER	the user.
MBT_TARGET_MVSMF_PASSWORD	the password.
MBT_TARGET_HLQ, **MBT_TARGET_VOLUME**	as **hlq** and **volume** in the file.
MBT_TARGET_JOBCLASS, **MBT_TARGET_MSGCLASS**	as **jobclass** and **msgclass**.

The variables **MBT_TARGET_HERCULES_*** and **MBT_TARGET_TN3270_*** add the other ways in the same manner; the *MBT Reference* lists them.

Operator Commands

mbt target console sends an operator command to a target and shows the reply:

```
mbt target console lab -- D T
```

The command goes through mvsMF, and, when mvsMF certainly did not take it, because there was no connection or it answered with an HTTP error, through the Hercules web interface. **[target.***name***.console]** changes the order. A command whose delivery is uncertain, because it was sent and no **order**
answer came, is not sent a second time: it may have run. A reply from MVS that rejects the command

is an answer too, and is not sent elsewhere. The project's Lua code can issue operator commands the same way (Chapter 11, "Extending MBT" on page 53).

Note: mvsMF does not check whether a user may issue operator commands: any user who can log on to it can issue any command through it. Keep that in mind when you decide who may reach mvsMF.

Moving from .env

MBT 2 kept the system in a file **`.env`** in each project. MBT 3 still reads it, and the variables **`MBT_MVS_*`**, when no target is defined, and warns on every run. **`mbt target import`** turns such a file into a target:

```
mbt target import .env --name lab
```

It writes the target into **`~/.mbt/targets.toml`**, makes it the default if it is the first, and makes the file readable by you alone. What the **`.env`** leaves out, such as the port (default 1080) or the job classes, it takes from where MBT 2 took it, and prints each value with its source. A password is copied only from the **`.env`** itself, never from the environment. An existing target of the same name is not replaced. Afterwards the project's **`.env`** can be deleted.

Chapter 7. Deploying

To try a program on MVS, its load modules have to be in a load library there. **`mbt deploy`** puts them into the project's development library on a target (Chapter 6, "MVS Systems: Targets" on page 25), replacing the members of an earlier deploy, also while a running server holds the library. This chapter describes which library that is and why it is a library of its own, what a deploy does, and how to keep the library from filling up.

The Development Library

A product installed with SMP (Chapter 9, "Packaging and SMP Installation" on page 39) lives in its target library, such as **`SUMS.LINKLIB`**, and that library belongs to SMP: SMP records which level of each module is in it. A deploy into it would leave SMP describing a level that is not on disk, without any way to notice. So a project deploys into a library of its own, by default *`NAME`***`.DEV.LINKLIB`**:

Table 5. Three libraries, three owners

Library	Owner	Filled by
*`NAME`***`.LINKLIB`**	SMP	the installation job, and only that
*`NAME`***`.DEV.LINKLIB`**	development	**`mbt deploy`**
*`HLQ`***`.`***`NAME`***`.`***`VRM`***`.TESTLIB`**	tests	**`mbt test --mvs`** ("On MVS: mbt test –mvs" on page 34); **`[deploy] test_target`** names another

`[deploy] target` names another library. A project whose name is longer than eight characters, and so is no qualifier, must name one: MBT does not shorten the name of a library a started task has to name in its JCL. **`--linklib`** *`DSN`* deploys into another library for one run.

The name carries no user's prefix on purpose. A development level is used by naming the library in the **`STEPLIB`** of a job or started task, and a procedure can only name a fixed data set.

Put the development library first in the `STEPLIB`, ahead of the SMP library. The first library that has a member wins, so a deploy overrides the installed level, and removing the DD statement returns to it. **Every library in an authorized `STEPLIB` must be APF-authorized**, the development library included: a single library that is not makes the whole task unauthorized, and the failure that follows does not name the cause. MVS 3.8j reads the APF list at IPL, so add the development library and the SMP library together.

A development library has no place on a production system. Sitting first in the **`STEPLIB`**, it runs a development build silently.

What a Deploy Does

`mbt deploy` first builds, as **`mbt build`** does: what it ships is the tree as it is now, stamped with the current commit, never what an earlier build left in **`build/`**. When everything is up to date, that step prints nothing. Then:

1. **Pack.** The modules go into one TSO TRANSMIT file of a load library.
2. **Check the room.** If the library exists, MBT reads how many tracks are free and estimates, generously, how many the modules need. When they do not fit, it stops before anything is copied ("When the Library Fills Up"). When it cannot measure the space, for a device or space unit it does not know, it warns and goes on without the check.
3. **Upload.** The TRANSMIT file goes to MVS through mvsMF, into a staging data set *hlq*`.MBT.XMIT.IN`.
4. **Receive and copy.** One job receives the file into a staging library, *hlq*`.`*PROJECT*`.MBTDPLY`, and copies its members with `IEBCOPY` into the development library, replacing members of the same name. The job opens the library `DISP=SHR`, so it works while a running server holds the library.
5. **Verify.** MBT lists the members of the library and checks that every deployed module is there. A job that ended cleanly but left a module out is an error.
6. **Report.** It shows how full the library is, with a warning from 80 per cent on, and removes the staging data sets.

Figure 23 shows the first deploy of a module into a library that did not exist yet. The project is a library project whose test module was deployed; its name, cut to eight characters, names the staging library.

```
$ mbt deploy
[mbt] Deploy target: IBMUSER.MBT.T105.LINKLIB
[mbt] Allocating IBMUSER.MBT.T105.LINKLIB (WORK00, room for several deploys, 20 directory blocks)...
[mbt] Uploading lstring370.deploy.xmit -> IBMUSER.MBT.XMIT.IN...
[mbt] RECEIVE IBMUSER.MBT.XMIT.IN -> IBMUSER.LSTRING3.MBTDPLY, IEBCOPY -> IBMUSER.MBT.T105.LINKLIB (replace, DISP=SHR)...
[mbt] IBMUSER.MBT.T105.LINKLIB: 23% of 30 tracks used, 1 extent(s)
[mbt] Deploy complete: 1 module(s) -> IBMUSER.MBT.T105.LINKLIB
```

Figure 23. A first deploy

A second deploy of the same module shows the same lines, without the allocation, and ends at 46 per cent of the 30 tracks: the old member's space is still there ("When the Library Fills Up").

`--module` *M* deploys only the named module (repeat it for more). `--dry-run` packs and stops: it needs no password and logs on to nothing.

A library that does not exist yet is allocated by the deploy job, with room for several deploys and a directory sized to its members and aliases (about three blocks for every ten names, 20 blocks at least), on the target's volume if the target names one (`volume`, "The File targets.toml" on page 25). MBT then checks that it landed on that volume, since an APF entry names a data set together with its volume. Without a volume, the system chooses one, and there is nothing to check.

A running server keeps the modules it loaded. A new member in the library reaches a program the next time it is loaded. A server that loaded its modules once at start-up runs the new level only after a restart.

When the Library Fills Up

Replacing a member does not free the space of the old one: in a partitioned data set, it stays dead until the library is compressed. Each deploy therefore uses more of the library, and the report after each deploy shows how much. When the next deploy would not fit, MBT stops with the numbers and the three ways out:

- **Compress the library** while nothing holds it, that is while the server is stopped. The plugin `mvslovers/mbt-stc` does it on the restart that activates a deploy anyway (below).
- `mbt deploy --reallocate` deletes the library and allocates it again on the same volume, then deploys. The new library has room for several deploys of the current modules and is never smaller than the old one, so running it again does not make it grow. Its directory has the blocks the modules

and aliases call for, and at least as many as the old library's member names would need. Before the delete, a short job allocates the new size on the volume and releases it again: if the volume has no room, MBT stops there and the old library stays as it was. It works only while nothing holds the library: otherwise the delete fails, and MBT stops with "most likely a running server holds it – stop it first".

- **`mbt deploy --ignore-space`** copies all the same, for a library whose secondary extents will take it.

If a data set runs out of room anyway, because it has no secondary quantity left or the volume is full, MVS ends the step with an SD37 (or SB37, SE37). MBT reports that as "ran out of space", names the abend, the step and the data set, says what the abend means, and exits with return code 4. When the data set is the library, the copy step stopped midway: IEBCOPY replaces the members one by one, so the library may hold some new members next to the old ones. Compress it or use **`--reallocate`**, then deploy again; until a deploy completes, treat the library as a mix of old and new. When the data set is the staging copy that the deploy receives first, the library was not written, and MBT keeps the staging copy for a retry of the RECEIVE.

The free entries of the library's directory cannot be measured from the workstation; a library with many members can run out of directory blocks before it runs out of tracks. A library that MBT allocates gets a directory sized to the deploy, but it does not grow: one that took on more names later needs **`--reallocate`**.

Note: Libraries from earlier deploys. Earlier versions of MBT deleted the library on every deploy and received it afresh, sized to its content, so such a library is full or nearly so. Its first deploy now needs **`--reallocate`**, once, with the server stopped.

Compressing on the Restart

A server that loaded its modules at start-up runs a deploy only after a restart, and while it is down nothing holds its library. The plugin **`mvslovers/mbt-stc`** uses that moment:

```
[plugins]
"mvslovers/mbt-stc" = "^0.1"

mbt deploy
mbt run restart                # the started task named after the project
mbt run restart -- HTTPD       # another one
```

`mbt run restart` submits a job that compresses the deploy target (IEBCOPY) and asks for the library exclusively. The job waits (**`IEF099I`**) while the server holds the library. Then the command stops the server with **`P`**, and the job runs. The command then starts the server again with **`S`**, and the new server waits in the same way until the compress is done. MVS orders the three, so the command never waits itself. If the stop cannot be issued, the command cancels the job and fails. **`mbt run --dry-run restart`** shows the steps and touches nothing. Nothing is installed on MVS for it; the job needs a free initiator in the target's **`jobclass`** for those few seconds.

The job goes in before the stop because a server that carries mvsMF, such as HTTPD, takes the way to submit it down when it stops. The operator commands then go to the next console channel of the target ("Operator Commands" on page 27), so such a target needs **`[hercules]`**.

Note: Not in the server's own procedure. A compress step in the procedure that starts the server looks simpler, but a job step that asks for a data set exclusively holds it for the whole job. The library would stay locked for as long as the server runs: every deploy would wait for the server to stop, and fail on its timeout.

The plugin reads the library from **`ctx.project.deploy_target`** ("What the Code Can Reach" on page 54). An MBT that predates it, and a project with neither a **`[deploy] target`** nor load modules, get the answer that there is no deploy target.

When a Deploy Fails

A deploy that fails on MVS ends with return code 4 and says why, also when it stopped for lack of room before anything was uploaded. An unknown **`--module`** or a project without built modules ends with return code 2, before MVS is reached.

When the job ran but did not deliver, because a module is missing from the library afterwards or the library landed on another volume, the job's output is in **`build/deploy.spool`**, and the uploaded TRANSMIT file is left in *`hlq`***`.MBT.XMIT.IN`**, so the RECEIVE can be repeated.

A failed deploy is not all or nothing. **`IEBCOPY`** replaces one member after the other, so a job that fails part-way can leave some of the new modules in the library beside old ones. MBT says so; look at the spool before you use the library.

Chapter 8. Testing

MBT runs a project's tests in three places: on the workstation, which takes seconds and needs no MVS; on MVS, in batch and under TSO, as load modules; and in a TSO session at a 3270 terminal, for what only an interactive user sees. This chapter describes how tests are written and found, and how each of the three runs works.

What a Test Is

A test is a program. It returns 0 when everything it checked was right, and anything else when something was not. **That return code is all MBT relies on:** on the workstation it is the exit code, on MVS the condition code of the job step. A test may print what it likes.

The header **<mbtcheck.h>**, which comes with MBT, makes such a program short and its output uniform (Figure 5 on page 9):

CHECK(*cond***, *msg*)**	records one check and prints **PASS:** or **FAIL:** with *msg*.
CHECK_EQ(*got***, *want*, *msg*)**	the same for two numbers, printing both when they differ.
mbt_test_summary(*name***)**	prints the summary and returns 0 when every check passed, else 1.

MBT counts the **PASS:** and **FAIL:** lines for its report, but decides by the return code alone. One test is one source with one **main**, because the counters of **<mbtcheck.h>** belong to the source that includes it.

A test that returns 0 without printing a single **PASS:** or **FAIL:** line has passed, but nothing in its output says what it checked. MBT shows it as passed *with no assertions*: **no assertions** in its row on the workstation, **(none)** beside the condition code on MVS, and a warning at the end that names each such test. The last line then reads **all host tests passed,** *n* **without assertions** or **all test steps passed,** *n* **without assertions**. The return code of **mbt test** stays 0: a test that relies on its return code alone is legitimate. When the mark is a surprise, the test most likely skipped its work, as one does that runs only when given an argument it did not get ("On MVS: mbt test –mvs" on page 34).

Finding the Tests

Every **.c** and **.asm** file under **test/**, in any subdirectory, is a test, named after its file in upper case: **test/mvs/tstgctx.c** is **TSTGCTX**. The name becomes a member name and the **PGM=** of a job step, so it must be a valid member name: at most eight characters, letters, digits, **@**, **#** and **$**, not starting with a digit. A file whose name is not, and two files that would give the same test name, are errors that name the files. Exclude such a file, or give it a **[test.***NAME***]**.

A test links its own source, the private archive of **[internal]** ("Shared Code: [internal]" on page 16), the dependencies and the C library. **[tests]** gives what all tests share:

```
[tests]
rent    = false
reus    = false
exclude = ["test/helpers/fixture.c"]    # under test/, but not a test
```

Every test needs **rent** and **reus**, from **[tests]** or from its own table, as a load module does. **exclude** names files under **test/** that are not tests, such as helpers.

A Test That Differs

A table **[test.*NAME*]** is needed only where a test differs from the rule:

```
[test.TSTSHA]
sources = ["test/mvs/tstsha.c", "src/sha256i.c", "src/sha256u.c"]

[test.TREXXVL]
host = false                        # MVS only

[test.TISTSO]
parm = { batch = "0", tso = "1" }    # a different argument per run

[test.TSTLOAD]
fixtures = { SYSEXEC = ["test/fixtures/HELLO", "test/fixtures/EMPTY"] }
```

sources the test's sources, when it needs more than its own file. A source named here by another test is not a test of its own.

host **false**: do not run it on the workstation, for a test that uses services only MVS has.

mvs **false**: do not build or run it for MVS, for a test that needs files only the workstation has.

parm the argument of the test program, the same in both MVS runs, or **{ batch = ..., tso = ... }** for one each ("On MVS: mbt test –mvs").

fixtures data sets the test reads on MVS ("Fixtures" on page 35).

A test table also takes the keys of a load module, such as **entry**, **startup** and **ac** (Table 3 on page 15).

On the Workstation: mbt test

mbt test compiles every test with the C compiler of the workstation and runs it there: with C99 and MBT's warning options, then **[build] cflags**, then **[build.host] cflags**. A test written in portable C, as the ones above, runs on the workstation and on MVS from the same source. A test with a source that is not C, such as an assembler routine, runs on MVS only, unless **[build.host] replace** puts a C source in its place for the workstation.

```
mbt test                      # every test
mbt test --only TSTSHA        # one; repeat --only for more
mbt test -v                   # the commands and every line of output
```

When a test needs something on the workstation that MVS provides otherwise, **[build.host]** supplies it for the workstation build alone:

```
[build.host]
cflags  = ["-Wno-unused-parameter"]
sources = ["../lstring370/src/lstr#*.c"]          # extra sources
replace = { "asm/istso.asm" = "src/irx#env.c" }    # this one instead of that one
```

This is the fastest check there is: run it after every change.

On MVS: mbt test –mvs

mbt test --mvs builds the tests as load modules, sends them to the target (Chapter 6, "MVS Systems: Targets" on page 25) and runs them there:

1. The test modules go into a library of their own, *hlq*.*PROJECT*.*VvRrMm*.**TESTLIB**, such as **IBMUSER.SUMS.V1R0M0D.TESTLIB** for version 1.0.0-dev. Tests never go into the library the project deploys to. **[deploy] test_target** names another one.

2. MBT writes a job with two steps for each test: one runs the program in batch (**`EXEC PGM=`**), the other under TSO (**`IKJEFT01`** with **`CALL`**). The development library of the project (**`[deploy] target`**, Chapter 7, “Deploying” on page 29), or the library **`--linklib`** names, follows the test library in the **`STEPLIB`**, so a test can load the project’s modules. When that library does not exist yet, the tests run from the test library alone, and MBT says so.
3. It submits the job, reads its output and prints a table: one row per test, one column for the batch run and one for the TSO run, each with its condition code.

Both runs exist because a program can behave differently under TSO. A test passes only when both end with 0. **`--only`** *`NAME`* runs a subset, **`--no-deploy`** reuses the test library already on MVS, and **`--target`** names the system.

Arguments. **`parm`** becomes the **`PARM=`** of the batch step and the argument of the **`CALL`** under TSO, and reaches **`main(argc, argv)`** in both. A test whose right answer depends on where it runs gets the expected answer this way: **`{ batch = "0", tso = "1" }`** tells it whether to expect TSO. MBT passes no argument of its own, so a test that does its work only when given one, such as **`PARM='RUN'`**, needs **`parm = "RUN"`** in its **`[test.`***`NAME`***`]`**. Without it the test skips itself, ends with 0, and is shown as passed with no assertions (“What a Test Is” on page 33). An argument that does not fit on the step’s card is continued on the next one; the longest is 49 characters for the batch step and 69 for the TSO step, enough for a data set name. **`mbt test --mvs`** refuses a longer one before it uploads anything, and names the test.

Authorized tests. A test that must run APF-authorized needs **`ac = 1`** in its **`[test.`***`NAME`***`]`** as well as libraries that are APF-authorized. A library on the APF list is not enough by itself: a module without **`ac = 1`** runs unauthorized from it, **`__isauth()`** returns 0, and the test fails in a way that looks like a wrong answer, not like a wrong attribute.

Fixtures

A test that reads members of a partitioned data set on MVS declares them as fixtures: for each DD name, the files that become its members.

```
fixtures = { SYSEXEC = ["test/fixtures/HELLO"], SYSPROC = ["test/fixtures/OLD"] }
```

Before each run, MBT allocates one partitioned data set per test and DD name, *`hlq`***`.`***`PROJECT`***`.FIX.`***`TEST`***`.DD`**, sized to what it will hold, loads each file as a member named after the file in upper case, and adds the DD statement to both steps of the test. A member is visible only under the DD that declares it, so a test of a search order can give one DD a member the other lacks.

An empty line of a file is loaded as a line of blanks, the same record in an 80-byte member; the load would otherwise drop it without a word.

When the Run Itself Fails

A test that fails and a run that could not test anything are different things, and the return code of **`mbt test --mvs`** tells them apart:

Table 6. Return codes of mbt test –mvs

Code	Meaning
0	every test passed in both runs.
1	at least one test failed.
2	the configuration is wrong: no test modules built, a fixture that is wrong or whose file cannot be read.

4	the job could not run or its output could not be read: the submit failed, JES rejected it, the wait for it ran out, or mvsMF could not return its output. MBT shows the reason, and the generated job is in **`build/test-runner.jcl`**.
5	a fixture could not be loaded, for example because its data set ran out of space. The table is shown, but those tests did not run against what they declared. 5 is reported before 1.

When only part of the output could be read, the steps whose result is missing show **`??`** and count neither as passed nor as failed.

Everything: mbt check

`mbt check` runs the tests on the workstation and then on MVS. When the first run fails, the second does not start.

Interactive Tests: mbt test –tso

Some things can only be tested the way a user meets them: a full-screen dialog, a TSO command's prompts, the messages at the terminal. For these, MBT runs a TSO session of its own over TN3270, as the user of the target's **`[tn3270]`** (Chapter 6, "MVS Systems: Targets" on page 25), and drives it from Lua.

Each file **`test/tso/*.lua`** is one such test. It returns a function that receives the session:

```
-- test/tso/time.lua
return function(t)
  t:logon()
  t:type("TIME"):enter()
  t:expect("TIME-")
  t:logoff()
end
```

`t:logon()`, **`t:logoff()`**	log on as the target's user, and off.
`t:type(`*`s`***`)`**	type *s* at the cursor; it returns the session, so **`:enter()`** can follow.
`t:enter()`, **`t:pf(`***`n`***`)`**, **`t:pa(`***`n`***`)`**, **`t:clear()`**, **`t:tab()`**	the keys. Each waits for the keyboard; **`t:enter{ wait = false }`** sends and returns at once, for a program that keeps the keyboard locked while it runs.
`t:expect(`*`text`*** **`, { timeout =`** *`s`* **`})`**	wait until *`text`* is on the screen; if it does not come, the test fails and shows the screen it saw.
`t:screen()`, **`t:text()`**, **`t:cursor()`**	read the screen and the cursor position.
`t:color(`*`row, col`***`)`**	the colour at a position, counted from 1: a name such as **`"red"`** or **`"green"`**.
`t.testlib`	the name of the test library **`mbt test --mvs`** uses, for a session that calls one of the tests.

A program can mark what it shows with colour: an error in red, an answer in green. **`t:color`** reads the colour of a position, so that a test can check it. The row and the column are the ones of **`t:cursor()`**. On MVS/CE, the lines TSO writes are red and the command a user types is green:

```
1  -- test/tso/colour.lua
```

```
 2  return function(t)
 3    t:logon()
 4    t:type("TIME"):enter()
 5    t:expect("TIME-")
 6    local row
 7    for r, line in ipairs(t:screen()) do
 8      if line:find("TIME-", 1, true) then row = r end
 9    end
10    assert(t:color(row, 2) == "red", "the answer is not red")
11    assert(t:color(row - 1, 1) == "green", "the command is not green")
12    t:logoff()
13  end
```

Figure 24. A test that checks colours

```
$ mbt test --tso
[mbt] 1 interactive test(s) on lab (lab:3270 as IBMUSER)

  TEST       TSO
  ---------- ----
  COLOUR     ok

[mbt] all 1 interactive test(s) passed
```

Figure 25. Running it

The test finds the row of the answer to **`TIME`** on the screen, and asks for the colour of its first character and of the command above it. A failed **`assert`** fails the test with the message and the line. The names, and how MBT arrives at the colour of a position, are in *MBT Reference*, “The TSO Test API”.

Each file gets its own logon, and the session is logged off whatever happens. **`mbt test --tso`** does not deploy the test modules itself: run **`mbt test --mvs`** first when a session needs them. Data sets a session works on – an exec, a CLIST, input – come from the repository: **`[tso.`***`NAME`***`] fixtures`** in **`mbt.toml`** names them by key, MBT loads them before the session, and the test finds their names in **`t.fixtures`** (*MBT Reference*, “The TSO Test API”).

Right after a system starts, mvsMF can answer before TSO takes logons: HTTPD starts before VTAM and TCAS are done, and on MVS/CE the logon works a second or two later. So **`mbt target ping --wait`** is no proof that **`mbt test --tso`** can start. **`mbt target ping --tso --wait`** *`seconds`* waits for both under one deadline: it logs on as the target’s **`[tn3270]`** user up to **`READY`** and off again, with the same TN3270 client the tests use.

The terminal is a 24x80 model 2 unless the target’s **`[tn3270]`** or the test’s **`[tso.`***`NAME`***`]`** gives a **`size`**, columns first as x3270 writes it: **`"80x43"`** for a model 4, **`"160x62"`** for an oversize screen. The logon and TSO line mode stay on 24x80; a full-screen dialog that switches to the larger screen shows all its rows in **`t:screen()`**.

```
[tso.ISPFMENU]
size = "160x62"
```

Chapter 9. Packaging and SMP Installation

A release of a project is a set of files: the load modules for MVS, the library for other projects, and, for a product that is installed on MVS, an installation package for SMP, the System Modification Program of MVS 3.8j. **`mbt package`** writes all of them into **`dist/`**. This chapter describes those files, how a project describes its SMP package, how its function identifier is chosen, and what an operator should check after an installation.

What mbt package Writes

`mbt package` builds the project, load modules and library both, and then writes the release files. Figure 26 shows it for the project **`sums`** of Chapter 4, “Building” on page 11, with the SMP package described below.

```
$ mbt package
[cc370] app/sums.c
[cc370] src/mean.c
[as370] asm/addup.asm
[ar370] sumsint.a (2 objects)
[ar370] sums.a (2 objects)
[ld370] SUMS (entry=@@CRT0)
[mbt] Packaging sums-1.0.0-dev-load.xmit (1 module(s) -> SUMS.V1R0M0D.LINKLIB)
[mbt] Packaging sums-1.0.0-dev-lib.tar.gz
[mbt] WARNING: [distribution.smp] target 'SUMS.LINKLIB' is addressed by the DD name LINKLIB, a DD of the SMP procedures (SMPAPP on MVS/CE
and TK5). SMP stores the DD name with every element it installs, so a later run of the standard procedures resolves it to the system
library of that name, not to 'SUMS.LINKLIB'. Set target_dd to a name of your own.
[mbt] Generated sums-1.0.0-dev-alloc.jcl, sums-1.0.0-dev-inst.jcl and sums-1.0.0-dev-accept.jcl (FMID TTST100) in build/dist-stage/
[mbt] Packaged sums-1.0.0-dev-dist.zip and sums-1.0.0-dev-dist.tar.gz (5 files)
[mbt] Package complete -> dist/
```

Figure 26. Packaging a release

name-*version*-**load.xmit**	the load modules, as a TSO TRANSMIT file of a load library.
name-*version*-**lib.tar.gz**	the library and its headers, in **`lib/`** and **`include/`**: what **`mbt deps`** of another project fetches (Chapter 5, “Dependencies” on page 19).
name-*version*-**dist.zip**, **.tar.gz**	the installation package for an operator to unpack: the load modules, one TRANSMIT file per text library (*name*-*version*-*qualifier*.**xmit**), the jobs that install them, the README, and the **`extra`** files under their own names.

The library archive is written only for a project with **`[lib]`**, the installation package only for one with **`[distribution]`**. A project with **`[distribution]`** but no load module to ship is an error: there is nothing to install (“Modules That Stay Out of the Package” on page 40).

Describing the Package

Two tables turn **`mbt package`** into a product installation:

```
[distribution]
readme = "README.md"

[smp]
fmid = "TTST100"              # a throwaway id: 1.0.0-dev is a prerelease
delete = []                 # nothing to replace
```

Figure 27. The installation package of sums

[distribution] says what goes into the package besides the load modules:

readme	the file the operator reads first. It is shipped as **README.md**.
extra	further files to ship as they are, such as a disk image the project builds.
[distribution.library.*dir***]**	a library of text members made from the directory *dir* of the project, such as sample JCL and procedures. It travels as a TRANSMIT file of its own and is received with TSO **RECEIVE**, outside SMP, into **target**, by default ***NAME*.*DIR***. With **element = "MAC"** or **"SRC"** SMP installs it instead, one element per member (“Libraries Installed by SMP” on page 43).

[smp] describes the SMP side: above all the function identifier, and the data sets the modules pass through. The **fmid** of Figure 27 on page 39 is set by hand because **sums** is at **1.0.0-dev**, a prerelease: MBT gives a prerelease no identifier of its own (“Prereleases and Throwaway Identifiers” on page 45).

DD names. The line **WARNING** in Figure 26 on page 39 is about the DD name of the target library. SMP addresses each library by a DD name and stores it with every element it installs, and MBT uses the last qualifier of the data set name. The default target ***NAME*.LINKLIB** is therefore known as **LINKLIB**, the name of **SYS1.LINKLIB** in the SMP procedures. The installation works, but a later run of the standard procedures resolves that name to the system library. Give each library a DD name of its own with **target_dd**, **distlib_dd** and **lklib_dd** in **[smp]**, and with **target_dd** and **distlib_dd** in a library with **element**. Set them before the first installation: the elements SMP has already recorded stay under the old name (*MBT Reference*, “DD Names”).

Modules That Stay Out of the Package

Every **[module.***NAME***]** goes into the package. Some modules should not: a test tool that a TSO test (“Interactive Tests: mbt test –tso” on page 36) calls, for instance, which belongs in the development library next to the product’s own modules but must not reach the library SMP owns. Write **ship = false** in its table:

```
[module.TESTTOOL]
sources = ["test/tool/*.c"]
rent    = false
reus    = false
ship    = false
```

MBT builds the module and **mbt deploy** puts it into the development library, as for any other module; **mbt package** and **mbt dist** leave it out of the load TRANSMIT file and out of the SYSMOD. The default is **true**. A value that is not **true** or **false** stops MBT with a configuration error.

A module with **ship = false** does not count as a module to ship. A project whose **[distribution]** has none left is refused: **mbt package** stops with return code 2 and says that the table needs at least one module to ship. A project without **[distribution]** and with no module to ship gets no load TRANSMIT file.

A Module for a System Library

Some modules do not belong in the product's library at all: an SVC routine is loaded from **SYS1.LPALIB** at the IPL. Give such a module a table **smp** (*MBT Reference*, "[module.NAME.smp]"):

```
[module.IGX00011]
entry   = "IGX00011"
startup = false
sources = ["asm/igx00011.asm"]
rent    = true
reus    = true
refr    = true

[module.IGX00011.smp]
syslib  = "LPALIB"
distlib = "AOST4"
source  = true
```

SMP then owns the module where it runs: the SYSMOD's JCLIN link-edits it into **SYS1.LPALIB**, and the **ACCEPT** keeps it in **SYS1.AOST4**, as IBM's own modules are kept. A copy job after every installation is no longer needed, and an upgrade replaces the module there (a **RESTORE** of such a module has not been measured).

The module travels as its source, not as a load module. The installation job assembles it with the target system's own IFOX00 and macros, so the user gets the source of the code that runs in supervisor state, and the system's own mappings of its control blocks go into it. It must therefore be a single **.asm** file that IFOX00 assembles with return code 0. **mbt build** and **mbt deploy** build it as before, for testing.

Two return codes change: the **APPLY** and the **ACCEPT** end with 4, because the module is not yet in the library SMP links it into (*MBT Reference*, "A Module in a System Library"). The module takes effect at the next IPL with **CLPA**, which the operator does; say so in the product's installation guide.

Note: A copy of the module already in the library. Take it out before the first **APPLY** (not measured as a sequence; derived from the first installation, where nothing is there to include). If **SYS1.LPALIB** holds a copy of the module that SMP did not put there (one copied by hand for an earlier release, say), the **APPLY** links that copy into the new module. Measured on TK5 with a copy that was not refreshable: the module in the library is reentrant and reusable but **not** refreshable, although the **PARM** says **REFR**. SMP does not report it, and the **ACDS** still records **REFR**. Check the attributes of the module in the library itself, with **IEHLIST**, or in the listing of the link edit (*MBT Reference*, "A Module in a System Library").

The Installation Jobs

The package holds three jobs, or two with **[smp] accept_fmid = false**. The operator uploads the TRANSMIT files to MVS in binary, as files of 80-byte records, runs the allocation job once (a second run ends with an error, "Submitting the Jobs Again" on page 47), puts the names of the uploaded files into the installation job, and submits it. The installation job ends after the **APPLY**. The operator tries the product, and submits the accept job when it is to stay.

The allocation job, *name*-*version*-**alloc.jcl**, allocates the target library and the distribution library, by default *NAME*.**LINKLIB** and *NAME*.**A***xxxx***LOD**. It has no step that deletes them, on purpose: after an installation the distribution library holds what SMP accepted, and scratching it would leave SMP reporting a level that is no longer there.

The job puts every library on a volume you name. Its two symbolic parameters, **TGTVOL** for the target libraries and **DLBVOL** for the distribution libraries, are set on the **PROC** statement at the top of the job, and default to the volumes of an MVS/CE system, **MVSRES** and **SMP000**. **On TK5, set TGTVOL=TK5RES and DLBVOL=TK5DLB.** The defaults can be changed for every package of a project with **target_volume** and **dlib_volume** in **[smp]**. A volume the system does not have does not make the job fail: it waits, and the

console shows **IEF238D** and asks for a device name or **CANCEL**. Reply **CANCEL**, correct the volume, and submit the job again (*MBT Reference*, "The Volumes of the Allocation Job").

A load library that must be APF-authorized goes on the volume of its APF entry. An entry in **SYS1.PARMLIB(IEAAPF00)** names a data set and a volume, **dsname volser**. The library is authorized only on that volume. The same name on another volume is a library that exists, holds the modules, and is not authorized, and the installation does not say so: no step fails, and a list of the members shows the modules. The program that needs the authorization finds out when it runs. So read the list before the allocation job runs:

- **The system has an entry for the library.** Set **TGTVOL** to its volume, on the **PROC** statement or on the **EXEC** statement (**//ALLOC EXEC SMPALLOC,TGTVOL=WORK00**), or once for every package of the project with **target_volume** in **[smp]**. The job then allocates every target library there. A product whose libraries have entries on different volumes writes the volume of the one that differs into its **DD** statement in place of **&TGTVOL**. If that leaves a symbol on the **PROC** statement that no **DD** uses, remove it ("Upgrading an Installation" on page 45).
- **The system has no entry yet.** Add one for the volume the job puts the library on, **TGTVOL**. MVS 3.8j reads the list at IPL, so the entry counts after the next IPL.

The installation job, *name*-*version*-**inst.jcl**, installs the product. The SMP steps run in order, and each is skipped when the step it depends on failed:

Table 7. The steps of the installation job

Step	What it does
DELOLD	deletes every data set the job receives into, the staging library and each text library, since TSO **RECEIVE** does not write into an existing data set.
RECV1...	receives each uploaded TRANSMIT file: the load modules into the staging library, by default *NAME*.*xxxx***LOAD**, and each text library into its target.
RECV	receives the SYSMOD into SMP. It is part of the job, not a file of its own.
APPLYCHK	checks the installation without changing anything.
APPLY	copies the modules into the target library, and the elements of an element library into theirs.

The job ends there, with return code 0 on a clean run. It leaves the staging libraries in place, and its closing comment says how to go on. With **[smp] accept_fmid = false** there is no accept job: the job holds a **CLEANUP** step behind the **APPLY** that deletes the staging libraries, and the level is never accepted.

The accept job, *name*-*version*-**accept.jcl**, is the decision to keep the level. Until it has run, a **RESTORE** can take the function out of the system again. Its **ACCEPT** copies the modules into the distribution library, the level a later **RESTORE** returns to, and cannot be undone. Submit it after an installation job whose **APPLY CHECK** and **APPLY** ended with return code 0, or with 4 for **HMA2461** (the upgrade of a predecessor) or **HMA4441** (a library of **SRC** elements) alone, not after 8 or 12. Its first step, **GUARD**, looks for the SYSMOD in the target zone. Only when it is applied do the **ACCEPT** and the **CLEANUP** run; otherwise the job ends with return code 8 in the step **NOTAPPL** and nothing is accepted or scratched:

```
IEF142I QBTSTINS DELOLD - STEP WAS EXECUTED - COND CODE 0000
IEF142I QBTSTINS RECV1 - STEP WAS EXECUTED - COND CODE 0000
IEF142I QBTSTINS HMASMP RECV - STEP WAS EXECUTED - COND CODE 0000
IEF142I QBTSTINS HMASMP APPLYCHK - STEP WAS EXECUTED - COND CODE 0012
IEF202I QBTSTINS HMASMP APPLY - STEP WAS NOT RUN BECAUSE OF CONDITION CODES
HMA3590     --- UZZ0001 REQ

IEF142I QBTSTACC HMASMP GUARD - STEP WAS EXECUTED - COND CODE 0004
```

```
IEF142I QBTSTACC NOTAPPL - STEP WAS EXECUTED - COND CODE 0008
IEF202I QBTSTACC HMASMP ACCEPT - STEP WAS NOT RUN BECAUSE OF CONDITION CODES
IEF202I QBTSTACC CLEANUP - STEP WAS NOT RUN BECAUSE OF CONDITION CODES
```

Figure 28. The accept job after an installation job that ended with 12 at the **APPLYCHK**

The same happens when the installation job never ran, and after a **RESTORE** has taken the function out. Behind a successful **ACCEPT** the **CLEANUP** deletes the staging library, and those of the element libraries. A failed **ACCEPT** leaves them in place, and the job can be submitted again. It needs no editing.

The **GUARD** tests for the entry of the SYSMOD in the target zone, not for its status text. A **RESTORE** that failed, for lack of the DD cards of the **APPLY**, leaves the entry and the status **REC APP RES ERR**, and the **GUARD** lets the job go on. Read the **RESTORE** output before you submit the accept job.

Someone who never accepts keeps the staging libraries. The closing comment of the installation job has the **DELETE** statements for them, to run with IDCAMS.

The SYSMOD is a **++FUNCTION** with one **++MOD** per load module:

```
++FUNCTION(TTST100) .
++VER(Z038)
   /* sums 1.0.0-dev                                          */
   /* Load modules are copied from the LKLIB, not re-bound.   */ .
++JCLIN .
//TTST100 JOB 1,'SUMS JCLIN',MSGLEVEL=1,CLASS=A
//COPYLOAD EXEC PGM=IEBCOPY
//ASUMSLOD DD  DISP=SHR,DSN=SUMS.ASUMSLOD
//LINKLIB  DD  DISP=SHR,DSN=SUMS.LINKLIB
//SYSIN    DD  *
  COPY INDD=ASUMSLOD,OUTDD=LINKLIB
  SELECT MEMBER=(SUMS)
/*
++MOD(SUMS) LKLIB(SUMSLOAD) DISTLIB(ASUMSLOD) .
```

Figure 29. The SYSMOD of sums 1.0.0-dev

Its **++JCLIN** is a copy job, not a link-edit: SMP copies each module as MBT linked it and does not bind it again. So every attribute set at the link, such as **ac = 1**, the entry point and the aliases, survives the installation. Aliases are carried as such on the **++MOD**.

Libraries Installed by SMP

A library of members that SMP should know, such as REXX functions and their sources, goes into the package as **[distribution.library.***dir***]** with **element = "MAC"** (or **"SRC"** for sources). Every file of the directory becomes a **++MAC** (or **++SRC**) of the SYSMOD, and the jobs read each from the staging library of its library through a **TXLIB** card of its own, in four steps. It then has about as many DD cards per step as the library has members: 235 for libraries of 114 and 121 members ran on MVS/CE and TK5.

The members are fixed 80, and nothing in the 80 columns is changed. **mbt package** refuses a file with a line past column 80, with a control character or a character that EBCDIC (CP037) has no equivalent for, and names the file, the line and the column. Sequence numbers in columns 73 to 80 are carried as they are.

Note: Before the first installation run the **LIST CDS** statements **mbt package** prints on the target system. A name SMP already has as the same type of element is silently not installed (**NOT SEL**, return code 0). A **SRC** with the name of a load module makes SMP assemble the member into that module, or end the **APPLY** at return code 12.

A later version that leaves a member out removes it from the target library and the distribution library when it installs, as for a load module. The Reference has the rules in full (*MBT Reference*, "Libraries as SMP Elements").

The Function Identifier

Every release that SMP installs is a SYSMOD with a function identifier, the FMID, seven characters: **`T`**, three letters for the product, and three digits for the version. **`[smp] prefix`** gives the first four, and MBT derives the rest from the version:

Table 8. FMIDs derived from the version

Version	`fmid`	`delete`
1.3.0	**`TSUM130`**	**`TSUM120`**
1.4.0	**`TSUM140`**	**`TSUM130`**
2.0.0	**`TSUM200`**	must be given

One per minor. The FMID is the release's major and minor version, and a **`0`**. A patch release is meant to become a PTF against its minor's FMID, which MBT does not build yet. Until it does, a patch release needs an explicit **`fmid`**, such as **`TSUM141`** for 1.4.1, **and an explicit `delete`**, here **`["TSUM140"]`**: a SYSMOD that replaces nothing does not own the modules it ships ("Checking an Installation" on page 47). MBT cannot tell a patch from a first level, so an explicit **`fmid`** without **`delete`** is an error; a product's first level writes **`delete = []`**. MBT refuses to spend the minor's identifier a second time.

Each release replaces its predecessor. **`delete`** names the FMID the release replaces. With a derived FMID it is the previous minor by default. A release x.0.0 has no previous minor to derive, so it names it, or writes **`delete = []`** for a product's first level, as **`sums`** does.

After a patch release, name it. When 1.4.1 was released as **`TSUM141`**, that SYSMOD owns the modules, and 1.5.0 must delete **`TSUM141`**, not the derived **`TSUM140`**. MBT refuses to package 1.5.0 with a derived **`delete`** when the repository has a tag of a 1.4 patch release, and asks for an explicit one.

No digit above 9. There is no room for two: at patch 9 the next release is a new minor, at minor 9 a new major.

When it is checked. A missing **`delete`** for x.0.0 or beside an explicit **`fmid`**, and a digit above 9, make the project file invalid: every command stops, even **`mbt build`**. A patch release without an explicit **`fmid`** stops **`mbt package`** and **`mbt release`** for a final version; a prerelease without an explicit **`fmid`** stops **`mbt package`** and **`mbt dist`** too ("Prereleases and Throwaway Identifiers" on page 45). The check for a released patch tag needs the repository's tags, which a shallow CI checkout does not have: it takes effect where a release is prepared, on the workstation.

Every FMID once. An identifier that was installed anywhere, even on a test system, is spent. Never give it to another release.

`T` keeps the identifiers out of IBM's, which on MVS 3.8j begin with **`E`**. Before a new prefix is taken into use, check on the systems it will be installed on that no SYSMOD of that name exists ("Checking an Installation" on page 47).

Prereleases and Throwaway Identifiers

A prerelease has no identifier of its own to derive. The identifier derived from **`1.4.0-dev`** is **`TSUM140`**, that of the release 1.4.0. Installing the prerelease build would spend it, and the release could no longer be installed under it on that system. The build would also own the modules, so that a patch release of the previous minor, shipped under an identifier of its own, installs nothing, with return code 0 (“Checking an Installation” on page 47). So **`mbt package`** refuses a prerelease whose **`[smp]`** gives no **`fmid`**. With the project of Figure 27 on page 39 and **`prefix = "TSUM"`** in place of the **`fmid`**:

```
$ mbt package
[cc370] app/sums.c
[cc370] src/mean.c
[as370] asm/addup.asm
[ar370] sumsint.a (2 objects)
[ar370] sums.a (2 objects)
[ld370] SUMS (entry=@@CRT0)
[mbt] Packaging sums-1.0.0-dev-load.xmit (1 module(s) -> SUMS.V1R0M0D.LINKLIB)
[mbt] Packaging sums-1.0.0-dev-lib.tar.gz
[mbt] ERROR: version 1.0.0-dev is a prerelease: the derived FMID TSUM100 is the release's, and an install of this build would spend it
and block the release and patch releases of the previous minor (NOT SEL) -- set [smp] fmid and delete explicitly, to a throwaway id
(Reference, SMP chapter)
```

Figure 30. A prerelease without a throwaway identifier

The load modules and the library archive are written, the installation package is not, and the return code is 2. **`mbt dist`** refuses the same way. A project with **`[distribution]`** that has no **`[smp]`** **`fmid`** must therefore set one before its first prerelease can be packaged: a **throwaway identifier**, one that no release will carry, and a **`delete`**, which an explicit **`fmid`** always needs. Figure 27 on page 39 shows both, with **`delete = []`** because the build replaces nothing.

Any seven characters. MBT checks that the identifier is a letter and six letters or digits, and no more. It does not check that it is not the release’s: **`fmid = "TSUM100"`** on **`1.0.0-dev`** is accepted, and has the consequences above. Choose one that no release will carry, for example under a prefix that no product uses.

Spent once installed. A throwaway identifier that was installed anywhere is spent like any other (“The Function Identifier” on page 44).

On a system that has the product. A SYSMOD that replaces nothing does not own the modules the product already has, so its elements come back **`NOT SEL`** with return code 0. Give the modules of the throwaway build names the product does not use, or install on a system without the product. Naming the product’s identifier in **`delete`** would delete the installed level’s modules.

Before the release. MBT does not notice a throwaway identifier at a release: **`1.0.0`** with an explicit **`fmid`** is packaged under it, without a warning. Remove **`fmid`** from **`[smp]`**, put the **`prefix`** back, and name the throwaway identifier in **`delete`** next to the previous minor’s, before **`mbt release`** (Chapter 10, “Releasing” on page 49). A **`delete`** that is set replaces the derived one. Without the throwaway identifier in it, a system that carries the throwaway build gets a release that installs nothing, or that stands beside the build, and no step fails; the Reference describes both (*MBT Reference*, “Replacing a Throwaway Installation”).

Upgrading an Installation

A release whose SYSMOD names its predecessor in **`delete`** upgrades an installation in one run of the installation job. SMP deletes the predecessor’s modules from the target library, copies the new ones

in, and records the new FMID as their owner. The **APPLY** then ends with return code 4, with the message **HMA2461** that the deleted SYSMOD has no backup: that is expected, and the job goes on. The **ACCEPT** of the accept job does the same in the distribution library.

The predecessor's identifier stays in the SMP inventory as a deleted SYSMOD: **LIST** shows it with **DELBY** naming the release that replaced it. It remains spent.

Text libraries shipped with **[distribution.library]** are received afresh on every installation, by design: a shipped library belongs to the product and is replaced with it. Changes made to it on the system are lost, so change a copy, in a library of your own.

A release that gives a library another DD name needs one more thing from the job. SMP finds the predecessor's elements by the DD name it recorded, so the job must define that name for the old data set. Without it the upgrade fails (**HMA2832**, return code 12), or, if the old name is **LINKLIB**, it silently deletes from **SYS1.LINKLIB** and leaves the old library populated. List the old libraries in the one release that moves them. The new level installs into a library of its own name, the old one stays where it is:

```
[smp]
target    = "ABC.ABCLINK"
target_dd = "ABCLINK"

[smp.predecessor]
LINKLIB = "ABC.LINKLIB"
```

MBT adds the cards to the installation job and to the accept job (*MBT Reference*, "Moving a Library to Another DD Name"). The release after that removes the table. Keeping the data set name and changing only the DD name works as well, and the line for the old DD name is required there too: the delete of the predecessor runs before the new modules are copied, so the new level's modules stay and the ones it no longer has are deleted.

With a library of its own name, the new library does not exist yet. The allocation job of the release would also allocate the libraries that do, and fails on them. Submit a job of your own instead, with one **IEFBR14** step and the **DD** statement of the new library taken from the allocation job, its volume written out.

If you cut the allocation job down instead, take two things out of it: the **DD** statements of the libraries that exist, and every symbol on the **PROC** statement that no remaining **DD** uses. MVS 3.8j does not ignore an unused symbol, it ends the job. The generated job of a product whose target library is new and whose distribution library exists has **PROC TGTVOL=MVSRES,DLBVOL=SMP000** on its **PROC** statement. The job cut down to the new target library keeps the one symbol its **DD** uses:

```
//SMPALLOC PROC TGTVOL=MVSRES
//ALLOC   EXEC PGM=IEFBR14
//ABCLINK  DD  DSN=ABC.ABCLINK,DISP=(NEW,CATLG,DELETE),
//            UNIT=SYSDA,VOL=SER=&TGTVOL,SPACE=(CYL,(5,2,20)),
//            DCB=(DSORG=PO,RECFM=U,BLKSIZE=15040)
//        PEND
//ALLOC   EXEC SMPALLOC
```

Left on the **PROC** statement, **DLBVOL** is a symbol that no statement uses, and the system rejects the job before it runs (Figure 31). It works the other way round as well: a job cut down to the distribution library alone loses **TGTVOL**.

```
13.28.57 JOB   34  IEF452I ABCBAD   JOB NOT RUN - JCL ERROR
    5     IEF653I SUBSTITUTION JCL - UNIT=SYSDA,VOL=SER=MVSRES,SPACE=(CYL,(5,2,20)),
    5     IEF657I SYMBOL NOT DEFINED IN PROCEDURE
```

Figure 31. An allocation job with a symbol that no DD uses

Two things an upgrade does not do:

- **It does not touch the predecessor's data sets.** SMP records the DD name a module was installed through, not the data set behind it. A product whose library names changed between two releases leaves the old library in place, populated and, if it was, APF-authorized. Changing the **STEPLIB** of the started task and the APF list to the new library, and scratching the old one, are then part of the upgrade, not tidying up: without them the old program keeps running.
- **It does not change the APF list.** An APF entry names a data set and its volume. A library that is on another volume than its entry says is not authorized ("The Installation Jobs" on page 41 describes how to put the new library on the volume of its entry).

A required version of another product does not belong into the SYSMOD. SMP's **REQ** demands that every SYSMOD it names is installed at once, so it can name only exactly one level of another product and has no way to say "this level or a later one". Leave **[smp] prereq** empty, its default, and say in the README which versions of other products the release needs.

A usermod the product cannot run without is the exception. Name it in **[smp] prereq** and the **APPLY CHECK** and the **APPLY** refuse to install without it. SMP checks the requisite again at **ACCEPT**, and a usermod that is applied but can never be accepted would fail that step on every system. So with **prereq** the **ACCEPT** of the accept job bypasses the requisite check, for the product's own SYSMOD only. The accept job ends with return code 4 and **---** *id* **REQ BYPASSED** for each prerequisite: that is expected, and the header of the job says so (*MBT Reference*, "A Prerequisite That Cannot Be Accepted").

Checking an Installation

Check every installation by the content of the target library, not by the return codes. SMP gives a module to the SYSMOD that owns its name. A SYSMOD that does not own a module it ships cannot install it: SMP marks the element **NOT SEL**, copies nothing, and every step still ends with return code 0. The message that shows a module was really installed is

```
HMA2380 COPY SUCCESSFUL - MOD=SUMS - LMOD=SUMS - LIBRARY=LINKLIB
```

and the check that cannot be fooled is a list of the members of the target library. After an upgrade of a server, check also that the server that is running is the new level, for example by the version it reports when it starts.

Whether an FMID is already known to a system is shown by **LIST**, with the zone named:

```
//LIST     EXEC SMPAPP
//SMPCNTL  DD  *
 LIST CDS SYSMOD(TSUM130) .
/*
```

Return code 4 and an empty list mean the identifier is free. A list that shows only **TYPE = FUNCTION** and **DELBY** is a deleted predecessor, which answers with return code 0 and is spent. **CDS** lists what is applied, **ACDS** what is accepted; check both.

Changing the Package Alone: mbt dist

mbt dist writes the installation package again from the load modules already in **dist/**, without building. It is for a change to the package itself, such as a new README or another data set name.

Submitting the Jobs Again

Both jobs end with an error when they are submitted a second time, and do not change the system. The listings come from a run on MVS/CE with the libraries and the SYSMOD of a BREXX/370 3.0.0 package. TK5 gives the same.

The allocation job starts with a step **CHECK** that looks every library it would allocate up in the catalog. If one is cataloged, the step ends with return code 8, the allocation step is not run, and the job ends with 8. The listing shows the libraries it found:

```
IEF142I BREXXALC CHECK - STEP WAS EXECUTED - COND CODE 0008
IEF202I BREXXALC ALLOC ALLOC - STEP WAS NOT RUN BECAUSE OF CONDITION CODES
NONVSAM ------- BREXX.LINKLIB
     IN-CAT --- UCPUB001
```

Figure 32. The allocation job, submitted again

The catalog is asked, not the volume. A second run with other volumes, for example after the defaults were changed, ends the same way. Delete the libraries first if they are to be allocated again, or take other names (*MBT Reference*, "The Allocation Job, Submitted Again").

The installation job ends with return code 8 when the SYSMOD is already received, because the **RECV** ends with 8 (**HMA3862**). Each SMP step names every SMP step before it, so the **APPLYCHK** and the **APPLY** are not run behind it:

```
IEF142I BREXXINS HMASMP RECV - STEP WAS EXECUTED - COND CODE 0008
IEF202I BREXXINS HMASMP APPLYCHK - STEP WAS NOT RUN BECAUSE OF CONDITION CODES
IEF202I BREXXINS HMASMP APPLY - STEP WAS NOT RUN BECAUSE OF CONDITION CODES
HMA3862 ** SYSMOD TBRX300 ALREADY RECEIVED
HMA2050    RECEIVE PROCESSING COMPLETED - HIGHEST RETURN CODE IS 08
```

Figure 33. The installation job, submitted a second time

The SYSMOD has to be taken out before the job is submitted again. A SYSMOD that is only received is taken out with an SMP step that has the DD cards of the **APPLY** and the statement **REJECT S(***fmid***) .**; one that is applied and not accepted, with **RESTORE S(***fmid***) .** in the same way. The job then runs through as on the first day.

Chapter 10. Releasing

A release of a project is a Git tag, **v***version*, and the files that the release workflow builds from it and publishes on the project's GitHub release page. MBT sets the version, commits and tags; the workflow builds and publishes. This chapter describes both, the prereleases that publish a version still in development, and the workflows that build every change.

Versions

The version is **`[project] version`** in **`mbt.toml`** and nowhere else. Between releases it ends in **`-dev`**: **`1.4.0-dev`** is the version being worked on that will become 1.4.0. A release removes the **`-dev`**, and the next development version follows it.

`[release] version_files` names further files in which the version is written, such as a header with the version as a string; MBT changes them together with **`mbt.toml`**. A program can also take its version from **`<buildstamp.h>`** ("The Build Stamp" on page 13), which needs no such file.

Releasing: mbt release

On a working tree at **`1.4.0-dev`**, with nothing uncommitted:

```
mbt release 1.4.0
```

`mbt release` does this, and stops at the first step that fails:

1. checks that the tree is clean and at a prerelease of 1.4.0, such as **`1.4.0-dev`** or **`1.4.0-rc1`**, that the tag **`v1.4.0`** exists neither in the repository nor on **`origin`**, and, for a patch release with an SMP package, that it has an explicit FMID ("The Function Identifier" on page 44);
2. sets the version to **`1.4.0`**, commits it as **`release: v1.4.0`**, tags the commit **`v1.4.0`**, and pushes the commit and the tag;
3. sets the version to the next development version, **`1.4.1-dev`** unless **`--next`** names another, commits it as **`chore: bump to 1.4.1-dev`** and pushes it.

Pushing the tag starts the release workflow ("The Workflows" on page 50). Nothing is undone when a step fails: a failure after the tag leaves the tag, and the next run says so and how to remove it. Every refusal and every failed Git step ends with return code 2. Use **`--next 1.5.0-dev`** when the next release is planned as a minor, so the tree does not carry a patch version that would need an FMID of its own. For a product with an SMP package, a next version with a number above 9, such as **`1.4.10-dev`** after 1.4.9, is refused before anything is tagged: release with **`--next 1.5.0-dev`** instead.

In a repository with more than one project, only the project whose project file is at the root may tag: the tags **v***version* of the repository are its tags. A repository with one project file, not at the root, lets that one tag; one with several and none at the root lets none. This holds for **`mbt prerelease`** as well.

Prereleases: mbt prerelease

`mbt prerelease` publishes the version in development without releasing it. On a clean tree, it tags the current commit with the current version, such as **`v1.4.0-dev`**, and pushes the tag. A version without **`-dev`** or **`-rc`***n* is refused: its tag is a release tag, which only **`mbt release`** sets. Run again later, it moves

the same tag to the newer commit, and the release workflow publishes the prerelease again. Nothing in the tree changes.

Another project can then depend on that prerelease with a range that names it, such as **>=1.4.0-dev** ("Version Ranges" on page 20). Because the tag moves, the archive behind it changes too, and the dependent project's **mbt deps** refuses the new one until **mbt deps --update** accepts it ("The Lock File" on page 21).

mbt prerelease itself does not package. The release workflow that the tag starts does ("The Workflows"), and for a product with an SMP package and no **[smp] fmid** it stops at the **Package** step: a prerelease gets no derived identifier, and nothing is published. Such a product sets a throwaway **fmid** and a **delete** before its first prerelease ("Prereleases and Throwaway Identifiers" on page 45), and takes the **fmid** out again before **mbt release**, with the throwaway identifier named in **delete**. A project without **[distribution]** is not affected.

The Workflows

MBT provides two reusable GitHub Actions workflows. A project's own workflow files only call them:

```
# .github/workflows/build.yml
name: Build
on:
  pull_request:
  push:
    branches: [main]
jobs:
  build:
    uses: mvslovers/mbt/.github/workflows/build3.yml@main
    with:
      host_tests: true          # also run mbt test

# .github/workflows/release.yml
name: Release
on:
  push:
    tags: ["v*"]
jobs:
  release:
    uses: mvslovers/mbt/.github/workflows/release3.yml@main
    permissions:
      contents: write
```

Both install MBT, which then switches to the version **[toolchain] mbt** pins.

build3.yml builds every pull request and every change of the main branch, against the current development state of CC/370 and LIBC/370, so that a change there that breaks the project shows early. With **host_tests: true** it also runs **mbt test**.

release3.yml builds a tag with the CC/370 and LIBC/370 that **[toolchain]** names, checks that the tag is **v** followed by **[project] version**, runs **mbt package** and publishes the files of **dist/** as the release. A tag with a hyphen, such as **-dev** or **-rc1**, becomes a prerelease, and a tag pushed again replaces its release. A **[toolchain]** entry that is a version means the tag **v***version* of that project, any other string a Git reference; a missing entry means the main branch, so a project without **[toolchain]** releases against the development state.

Both install maclib370 for the system macros. A project with COBOL sources also needs **cobc370**: **with: cobc370_ref:** *ref* builds it from that tag, branch or full commit SHA of **brazilofmux/cobc370** and puts it on **PATH** ("COBOL Sources" on page 14).

Pin **uses:** to a tag of MBT, such as **@v3.2.0**, instead of **@main**, so that workflow and tool do not drift apart.

A workflow that deploys or tests on MVS describes its target in environment variables from the repository's secrets ("Targets in CI" on page 27).

After the Release

The workflow publishes the files with notes that GitHub generates from the pull requests and commits since the last tag. Before you announce the release, rework the page for someone who does not know the project yet: what the release is, what a user notices, how to install it, and then the section of **`CHANGELOG.md`** for the version, which the workflow does not read.

Chapter 11. Extending MBT

Some projects need a step MBT does not know: a source generated from a table, a disk image that ships with the product, a server to restart after a deploy, files to upload. Such steps are written in Lua and run inside MBT, with a small set of functions that let them act without being able to change the build. This chapter shows where the Lua lives, the three kinds of extension, what the Lua code can reach, and how extensions are shared as plugins.

Where the Lua Lives

Table 9. Where extensions live

File	For
`mbt/init.lua`	the project: what it needs. Further modules go into **`mbt/lua/`**: **`require("webroot")`** loads **`mbt/lua/webroot.lua`**.
`~/.mbt/init.lua`	you: what only your workstation needs, such as a notice after a release.
a plugin	several projects: Lua that someone else maintains, declared in **`[plugins]`** (“Plugins” on page 55).

Only a project with **`mbt.toml`** loads Lua. Commit **`mbt/`**; it is part of the project. At one point of a run, plugins come first, then the project, then your own file.

Hooks, Tasks and Commands

A hook runs at a fixed point, before or after one of MBT’s commands:

Table 10. Hook points

Point	For example
`before_build`, **`after_build`**	generate a source; report module sizes
`before_test`, **`after_test`**	prepare test data; pass the results on. In both, **`ctx.kind`** says **`host`**, **`mvs`** or **`tso`**.
`before_package`, **`after_package`**	build a file that ships in the package
`before_dist`, **`after_dist`**	add to the installation package
`before_deploy`, **`after_deploy`**	stop a server, start it again
`before_release`, **`after_release`**	refuse a release without a changelog section; announce it
`on_failure`	collect what a failed command left

```
-- mbt/init.lua: restart the server after a deploy
mbt.hook("after_deploy", function(ctx)
  ctx.console("P HTTPD")
  ctx.console("S HTTPD")
end)
```

An error in a **before_** hook stops the command before it does anything. An error in an **after_** hook undoes nothing, a deploy cannot be taken back, but the command then fails: a deploy whose restart failed is not reported as a success.

A task declares its inputs and outputs. It runs before the commands it names, ahead of their **before_** hooks and at most once per run: the first time, when an output is missing or older than an input, when the task itself, a Lua file or **[tools]** changed, and every time if it has no outputs. Its outputs are removed before it runs and after it fails, and a task that did not write them fails. A task with inputs but no outputs, and an input that matches no file, are errors:

```
mbt.task {
  name    = "tables",
  before  = { "build" },
  inputs  = { "tools/tables.txt" },
  outputs = { "src/tables.c" },
  run = function(ctx)
    ctx.exec { "python3", "tools/gen.py", ctx.inputs[1], ctx.out[1] }
  end,
}
```

A command of the project's own runs with **mbt run** *NAME* **--** *args*:

```
mbt.command {
  name        = "upload",
  description = "upload the web image",
  run = function(ctx)
    ctx.exec { ctx.tool("ufsd-utils"), "upload", "build/web.img", "--dsn", "LAB.WEBROOT" }
  end,
}
```

mbt.command("upload", function(ctx) ... end) is the short form, without a description. **mbt run** alone lists the tasks and commands, with their descriptions where they have one. **mbt run** *NAME* runs a task even when it is up to date. **--dry-run** runs the Lua but starts no program and sends no request to MVS that changes anything; the code sees **ctx.dry_run**.

What the Code Can Reach

Every hook, task and command gets one table, **ctx**. The Lua code has no other way to the outside: no **os**, no **io**, no network of its own.

Table 11. The functions and values of ctx

Name	What it is
ctx.project	the project's name, version, modules and tests, and **deploy_target**, the library **mbt deploy** writes to, as a copy.
ctx.exec{*argv***}**	runs a program. The arguments are a list, and no shell is involved, so a value with blanks stays one argument. A non-zero exit code is an error unless **check = false** is given; **env = {...}** adds environment variables. Under **mbt run** the program writes to the terminal; in a hook its output is returned.
ctx.tool(*name***)**	the path of a tool from **[tools]**, fetched and checked if need be ("Host Tools" on page 22).
ctx.fs	**read**, **write**, **exists**, **mkdir**, **remove** and **list**, inside the project only.
ctx.target()	the selected MVS system: hosts, ports, users, never a password.
ctx.console(*cmd***)**	an operator command ("Operator Commands" on page 27); returns the reply lines and the channel that delivered it.

ctx.mvs.request{…}	a request to mvsMF in MBT's session: method, path, body; returns status, body and headers. With **auth** it goes with Basic, with no credentials or with a given token instead, for tests of the logon itself.
ctx.mvs.token()	the session token, to hand to another program in its environment.
ctx.mvs.session{ user }	a session of its own, as the target's user or as a further one (*MBT Reference*, "targets.toml"): **s:request{}**, **s:token()**, **s:logout()**.
ctx.out, **ctx.inputs**	in a task: its outputs, and the files its inputs matched.
ctx.result	in an **after_** hook: what the command produced; in **on_failure**: **command** and **error**.
ctx.args, **ctx.kind**, **ctx.dry_run**	the arguments of **mbt run**, the kind of test run, the dry run.
ctx.log, **ctx.warn**	write a line of MBT's output.

Each call into Lua runs with a limit on the work it may do and on its memory (256 MiB). Programs started with **ctx.exec** are not limited. **print** writes to MBT's output.

Extensions act, they do not change the build. They can write files, run programs, issue commands and stop a command. Compile options, sources and module attributes change only in **mbt.toml**, so a build can always be explained from the project file.

Plugins

Lua that several projects need is published as a plugin: a GitHub release of its own repository, declared and pinned like a dependency:

```
[plugins]
"mvslovers/mbt-ufs" = "^0.1"

local ufs = require("mvslovers/mbt-ufs")
ufs.webroot { from = "static", image = "build/webroot/web.img" }
```

mbt deps resolves the range, fetches the release's *repo*-*version*-**plugin.tar.gz**, records its checksum in **mbt.lock** and stages it in **.mbt/plugins/**. A build uses only what is staged. The archive holds **plugin.toml**, **init.lua** and **lua/**. **plugin.toml** names the version of the interface the plugin was written for, and the programs it may start: **ctx.exec** from a plugin's code runs nothing else. A plugin written for an interface MBT does not provide is refused with a message.

.mbt/deps.local.toml can point a plugin at a working copy, as it does for a dependency ("Building Against a Working Copy" on page 22).

mvslovers/mbt-stc ("Compressing on the Restart" on page 31) shows how far the functions of **ctx** reach. Its command **restart** reads the library from **ctx.project.deploy_target**, submits a job through **ctx.mvs.request**, and stops and starts the server with **ctx.console**. It starts no program on the workstation.

Appendix A. Migrating from MBT 2

This appendix takes a project from MBT 2 to MBT 3. The build does not change: the same CC/370 commands write the same object modules, archives and load modules. What changes is everything around it.

Table 12. MBT 2 and MBT 3

	MBT 2	**MBT 3**
The tool	a Git submodule **`mbt/`** and a **`Makefile`**	the program **`mbt`** on the **`PATH`**
Its version	the submodule's commit	**`[toolchain] mbt`**
Project file	**`project.toml`** and **`VERSION`**	**`mbt.toml`**
Commands	**`make`** *`target`*	**`mbt`** *`command`*
MVS systems	**`.env`** in each project	**`~/.mbt/targets.toml`**
Extra steps	rules in the **`Makefile`**	Lua, or a plugin
CI	**`build.yml`**, **`release.yml`**	**`build3.yml`**, **`release3.yml`**

The project should build with MBT 2 before it is migrated.

Converting the Project File

```
mbt migrate --dry-run     # print the new mbt.toml, change nothing
mbt migrate               # write mbt.toml, remove project.toml and VERSION
```

`mbt migrate` reads **`project.toml`** and writes **`mbt.toml`**. Comments travel with the key or table they stand above. Before it writes anything, it loads the new file and compares it with the old one; if they differ in anything but the intended changes, nothing is written. Where it drops a comment, or where the **`Makefile`** does more than include MBT, it says so on standard error. Then pin the version:

```
[toolchain]
mbt = "3.0"
```

What you will notice in the new file:

- **Modules are tables keyed by name**: **`[[module]] name = "UFSD"`** becomes **`[module.UFSD]`**. MBT orders modules by name wherever order shows, so the first build lists them in a different order. The load modules do not change.
- **`rent` and `reus` are stated on every module**; **`norent = true`** becomes **`rent = false`**. So does **`norent = false`** (and **`noreus = false`** becomes **`reus = false`**), with a **`NOTE`**: MBT 2 passed no flag for it, so the module was never reentrant, and the migration keeps the load module as it is. Write **`rent = true`** yourself if the module really is reentrant.
- **`[build]` is shorter.** **`include/`** is on the include path by convention, and **`-Wall -Wextra -Werror`** are MBT's own, so the migration leaves all three out. Other **`-I`** pairs move to **`include`**.
- **Tests are found** under **`test/`** ("Finding the Tests" on page 33). A test needs a table only where it differs. The migration keeps every source list that differs from the found default, so the test

modules stay the same, and puts a file under **`test/`** that MBT 2 did not build as a test into **`[tests] exclude`**.

- **The version is in `[project] version` only. `VERSION`** is gone.
- **The FMID is derived** from **`[smp] prefix`** and the version (“The Function Identifier” on page 44), when the old one equals the derived: for a version *x*.*y*.**0** with *y* > 0 and the predecessor as **`delete`**. Otherwise **`fmid`** and **`delete`** are kept as they were. An explicit **`fmid`** without **`delete`** gets **`delete = []`** beside it, with a note: MBT 2 read a missing **`delete`** as none, MBT 3 wants it said. If the release replaces an earlier one, name that FMID instead.
- **Tables are renamed**: **`[project] type`** is **`kind`**, **`[host]`** is **`[build.host]`**, **`[test_deploy] target`** is **`[deploy] test_target`**, **`[distribution.smp]`** is **`[smp]`**, **`[[distribution.library]]`** is **`[distribution.library.`***DIR*`]`**, **`parm_batch`** and **`parm_tso`** are **`parm`**, and **`[[test.fixture]]`** is **`fixtures`**. Values equal to their defaults, such as **`[deploy] target`** of *NAME*`.DEV.LINKLIB`, are left out with a note.
- **C99 is the default.** MBT 2 compiled C89 with GNU extensions unless the project said otherwise, MBT 3 compiles C99. The code generated changes, the meaning does not; run the project’s tests on MVS once. **`cflags = ["-std=gnu89"]`** keeps the old dialect.
- **`startup = "crt0"` and `"crt1"` are gone**: the C start-up comes out of **`libc.a`**, which needs LIBC/370 2.3.0 or later.

A directory with both **`mbt.toml`** and **`project.toml`** is an error.

Removing the Submodule

In the same change, remove the **`Makefile`** and the submodule with one command:

```
git rm Makefile mbt .gitmodules     # mbt was the only submodule
git rm Makefile mbt                 # there are others: keep .gitmodules
rm -rf .git/modules/mbt
git config --remove-section submodule.mbt
```

Look first for code of your own that imports from the submodule (**`git grep -n "mbt/scripts"`**): a test runner or install helper may use MBT 2′s Python classes, which go with it, and MBT 3 offers no Python library in their place. Such a tool becomes a command in **`mbt/init.lua`** that hands it **`ctx.target()`** and **`ctx.mvs.token()`** in its environment (“What the Code Can Reach” on page 54), and runs as **`mbt run`** *NAME*. The tool keeps a small mvsMF client of its own and fails on an empty token. A tool that logs on to TSO needs a password, which the Lua never sees; its place is **`mbt test --tso`**.

Keep **`.mbt/`**, **`build/`** and **`dist/`** in **`.gitignore`**. The directory **`mbt/`** is the project’s own from now on: it holds the project’s Lua (Chapter 11, “Extending MBT” on page 53), so do not ignore it. Commit **`mbt.lock`** as before.

Commands

Table 13. make targets and mbt commands

MBT 2	MBT 3
`make`	**`mbt build`**
`make all`	**`mbt build --all`**
`make` *name*	**`mbt build`** *NAME*
`make test`	**`mbt build --tests`**
`make test-host`	**`mbt test`**
`make test-mvs ARGS="--only X"`	**`mbt test --mvs --only X`**

`make check`	**`mbt check`**
`make deps`	**`mbt deps`**
`make package`, **`make dist`**	**`mbt package`**, **`mbt dist`**
`make deploy`	**`mbt deploy`**
`make doctor`, **`make compiledb`**	**`mbt doctor`**, **`mbt compiledb`**
`make clean`, **`make distclean`**	**`mbt clean`**, **`mbt distclean`**
`make release VERSION=`*v*	**`mbt release`** *v*
`make prerelease`	**`mbt prerelease`**
`VERBOSE=1 make`	**`-v`** on the command

Behaviour you may notice:

- **`mbt deps` never moves a pin by itself** (“The Lock File” on page 21). MBT 2 re-pinned a prerelease that was published again, with a warning. A lock written that way may name an archive that no longer exists: expect the first **`mbt deps`** of a project that depends on a **`-dev`** version to ask for **`--update`**.
- **`--target` names a system now**, and **`--linklib`** the library (“The Development Library” on page 29). The default library is *`NAME`***`.DEV.LINKLIB`**.
- **`mbt deploy` replaces members** instead of deleting and receiving the library (Chapter 7, “Deploying” on page 29).

MVS Systems

Turn the project’s **`.env`** into a target once per workstation, with **`mbt target import .env --name`** *`name`* (“Moving from .env” on page 28). Then delete **`.env.example`** from the project: it documents variables MBT 3 no longer reads.

Makefile Rules

A rule in the **`Makefile`** beyond MBT’s own targets becomes Lua in **`mbt/init.lua`**, a tool in **`[tools]`**, or a plugin (Chapter 11, “Extending MBT” on page 53). **`mbt migrate`** names the rules it found.

CI

Replace the two workflow files by the ones in “The Workflows” on page 50. Keep any other job of your own. Then look for anything of the project’s own that reads **`project.toml`**, such as a checking script, and for comments that describe how MBT 2 moved **`VERSION`** or the FMID.

Checking the Result

Build the same commit with MBT 2 in a second checkout and compare **`build/`** and **`dist/`**. Object modules from assembler sources that use **`&SYSDATE`** or **`&SYSTIME`** carry the date of the assembly, and load modules the time of the link; set the same values on both sides to compare byte for byte:

```
export ASMDATE=10/07/26 ASMTIME=12.00 LDDATE=26280 LDTIME=120000
```

A project whose **`project.toml`** gave no **`-std`** compiled C89 with MBT 2: for the comparison, add **`cflags = ["-std=gnu89"]`** on the MBT 3 side, and remove it afterwards. Archives (**`.tar.gz`**) differ anyway in how they are written; compare what they contain.

Appendix B. Glossary

This glossary defines the terms of MBT, and of MVS and SMP as far as this book uses them. A term printed in *italics* in a definition has an entry of its own. The terms of the toolchain itself are in the glossary of the *CC/370 User's Guide*.

accept job — The job *name*`-`*version*`-accept.jcl` of an SMP package. Its `ACCEPT` makes the installed level the one a `RESTORE` returns to, and its `CLEANUP` deletes the staging libraries. The user submits it after trying the product.

APF — The authorized program facility of MVS. A program runs authorized only if its *load module* has authorization code 1 and every library of an authorized `STEPLIB` concatenation is listed as APF-authorized, by data set name and volume.

build stamp — The header `<buildstamp.h>` that MBT writes for each build, with the project's name, version and commit.

deploy — Putting the load modules of a *project* into its *development library* on a *target*, with `mbt deploy`.

dependency — Another *project* whose *library* a project links with, declared in `[dependencies]` and pinned in the *lock file*.

development library — The load library a project deploys to, by default *NAME*`.DEV.LINKLIB`. It is never the library *SMP* installs into.

distribution library — The library in which *SMP* keeps the accepted level of each element, the level a `RESTORE` returns to.

element — A member that *SMP* installs and tracks by name, here a `++MAC` or `++SRC` made from a file of a library with `element` in `[distribution.library.`*dir*`]`. Each is read from the staging library through its own `TXLIB` card.

FMID — The function identifier of a *SYSMOD* of type `FUNCTION`: `T`, three letters for the product and three digits for the version, such as `TSUM140`.

fixture — A data set a test reads on MVS, loaded by `mbt test --mvs` from files of the project.

hook — Lua code that runs before or after one of MBT's commands.

launcher — The part of `mbt` that runs the version of MBT a project pins, fetching it if need be.

library — A static library of object modules, an archive, built from `[lib]` and published for other projects.

load module — An executable program in a load library on MVS, made by the linkage editor from object modules.

lock file — `mbt.lock`: the exact version and checksum of every *dependency*, *tool* and *plugin* of a project.

mvsMF — A REST interface on MVS, compatible in part with z/OSMF, through which MBT uploads files, submits jobs and reads their output.

plugin — Lua published by another repository, declared in `[plugins]` and pinned in the *lock file*.

prerelease A version with **`-dev`** or **`-rc`***n*, published before the release. A *dependency* range gets one only if it names one.

project A directory with an **`mbt.toml`**: what MBT builds.

PTF A program temporary fix: a *SYSMOD* that services a function. MBT does not build PTFs yet.

reentrant Said of a *load module* that one copy can serve several tasks at once, because it never changes itself. Declared with **`rent = true`**.

serially reusable Said of a *load module* that a copy can be used again once a task has finished with it. Declared with **`reus = true`**.

SMP The System Modification Program of MVS 3.8j, release 4, which installs products and records what it installed.

staging data set A data set that holds files on their way: a deploy uploads into *`HLQ`***`.MBT.XMIT.IN`** and receives into the staging library *`HLQ`***`.`***`PROJECT`***`.MBTDPLY`**; an installation receives into *`NAME`***`.`***`P4`***`LOAD`**.

SYSMOD A system modification: the unit SMP installs, such as a **`FUNCTION`** for a product release.

target An MVS system MBT deploys and tests on, described in **`~/.mbt/targets.toml`**.

task Lua code with declared inputs and outputs, run before a command when its outputs are out of date.

test library The library **`mbt test --mvs`** puts the test modules into, by default *`HLQ`***`.`***`NAME`***`.`***`VRM`***`.TESTLIB`**; **`[deploy] test_target`** names another.

tool A program for the workstation, fetched from a GitHub release and pinned like a *dependency*, declared in **`[tools]`**.

TRANSMIT file A file in the format of the TSO **`TRANSMIT`** command, which TSO **`RECEIVE`** turns back into a data set. MBT ships load libraries in it.

Index

T

--target 26

.

.env 28

[

[dependencies] 19
[distribution] 39
[internal] 16
[lib] 16
[plugins] 55
[smp] 39
[test.NAME] 34
[toolchain] 22
[tools] 22

A

ac 35
allocation job
  second run 47
APF 29, 35, 42, 46
assembler source 14

B

build directory 8
build stamp 13
build3.yml 50
buildstamp.h 13

C

CI
  target 27
cobc370 14
COBOL 14
compile options 13
compile_commands.json 17
compress 30
ctx 54

D

DD name 40
dependencies, assembler 12
dependency 19
  working copy 22
deploy
  first 30
  space 30
deps.local.toml 22
DEV.LINKLIB 29
development library 29
dist directory 39

E

element library 43
extension 53
  rules 55

F

first project 7
fixture 35
FMID 44
  throwaway 45

G

GitHub
  rate limit 4
GitHub Actions 50
GITHUB_TOKEN 4

H

hook 53
host build 1
host test 34

I

IEAAPF00 42
installation job 41
  second run 47

K

keychain 26

L

launcher 2
library 1
  building 16
load module 1
  declaring 15
LPALIB 41
Lua 53

M

max_rc 14
MBT 1
  installing 3
  release 3
  tasks 1
  version 2
MBT 2 2
  comparing 59
  migrating 57
mbt build 11
  first 8
  steps 11
mbt check 36

mbt clean 13
mbt compiledb 17
mbt deploy 29
  dry run 9
  errors 32
  steps 29
mbt deps 20
  --update 21
mbt dist 47
mbt distclean 13
mbt doctor 4
mbt migrate 57
mbt package 39
mbt prerelease 49
mbt release 49
mbt run 54
mbt target 26
  ping --tso 37
mbt target console 27
mbt target import 28
mbt test 34
  --mvs 34
  --tso 36
  return codes 35
mbt-stc 31
mbt.lock 21
mbt.toml
  first 7
MBT_HOME 4
MBT_TARGET_MVSMF_URL 27
mbtcheck.h 33
module-data check 16
mvsMF
  session 27

O

operator command 27

P

parm 35
password 26
plugin 55
predecessor 46
prerelease 49
  dependency 21
  SMP package 45
prerequisite 47
project 1
  kind 2
  layout 7

R

REFR 41
release 49
  repository 49
release3.yml 50
RENT 15
  writable data 16
REUS 15

S

semantic versioning 20
ship 40
SMP 39
  checking 47
  DELETE 45
  LIST 47
started task
  restart 31
startup 15
SYSMOD 43
system library 41

T

t:color 36
target 2, 25
targets.toml 25
task 54
test 33
  discovery 33
  first 9
  no assertions 33
  return code 33
test tool 40
TESTLIB 34
TN3270 36

V

version 49
version range 20

~

~/.mbt 4