USER MANUAL

Tatara and Tanren

Javier Lavandeira

Version 1.2.0

2026

https://tatara.tools

Tatara and Tanren User Manual
Version 1.2.0, 2026
`https://tatara.tools`

Copyright © 2026 Javier Lavandeira.

This manual is licensed under the Creative Commons Attribution 4.0 International License (CC BY 4.0): `https://creativecommons.org/licenses/by/4.0/`. The example programs it contains are licensed under the Apache License, Version 2.0: `https://www.apache.org/licenses/LICENSE-2.0`.

MSX is a trademark of MSX Licensing Corporation.

MACRO-80 (M80) and LINK-80 (L80) are products of Microsoft Corporation. Microsoft is a trademark of the Microsoft group of companies.

The AS assembler and the LD linker are copyright © Egor Voznesensky.

Nextor is copyright © Néstor Soriano.

All other product names mentioned in this manual, including CP/M, MSX-DOS, Z80 and R800, are trademarks or registered trademarks of their respective owners. They are used here only to identify the products, and their use does not imply any affiliation with or endorsement by their owners.

Contents

About this manual **xiii**

I Getting started **1**

1 Introduction **3**
1.1 What Tatara and Tanren are . . . 3
1.2 Where the names come from . . . 4
1.3 What they can do . . . 4
1.4 What you need . . . 6
1.5 What is next . . . 6

2 How an assembler and a linker work **7**
2.1 Two steps . . . 7
2.2 What the assembler does . . . 8
2.3 What the assembler cannot know . . . 9
2.4 What the linker does . . . 10
2.5 A worked example . . . 10
2.6 Code and data: segments . . . 13
2.7 Why two steps are worth it . . . 14
2.8 The words in this chapter . . . 14

3 Installation **17**
3.1 What you get . . . 17
3.2 Where the files go . . . 18
3.3 Installing, step by step . . . 19
3.4 Setting PATH and TATARA . . . 20
3.5 Checking the installation . . . 21
3.6 Updating . . . 22

4 A first program **23**
4.1 Before you start . . . 23
4.2 The program . . . 24
4.3 Assembling it . . . 25
4.4 Linking it . . . 26
4.5 Running it . . . 26
4.6 The build file . . . 27
4.7 A program in two modules . . . 27

4.8 What is next . . . 29

II The assembler, Tatara 31

5 Running TATARA 33
5.1 The command line . . . 33
5.2 Filenames . . . 34
5.3 The options . . . 36
5.4 What TATARA prints . . . 37
5.5 Listings . . . 38
5.6 When TATARA does not start . . . 39

6 Source format 43
6.1 Lines . . . 43
6.2 The four fields . . . 44
6.3 Where a label goes . . . 44
6.4 Comments . . . 46
6.5 Upper and lower case . . . 46
6.6 Spaces, tabs, line length and line endings . . . 46

7 Names 49
7.1 The characters of a name . . . 49
7.2 Length . . . 50
7.3 Upper and lower case . . . 51
7.4 Names that are hard to reach . . . 53

8 Expressions 55
8.1 Values are 16 bits . . . 55
8.2 Numbers . . . 56
8.3 Characters . . . 56
8.4 Names and the location counter . . . 57
8.5 Operators . . . 57
8.6 What kind of value . . . 59

9 Instructions 61
9.1 The instruction set . . . 61
9.2 Operands . . . 62
9.3 Relative jumps . . . 63
9.4 The R800 instructions . . . 64
9.5 The undocumented instructions . . . 65
9.6 When an operand is wrong . . . 68

10 Segments and the location counter 69
10.1 The location counter . . . 69
10.2 Code, data and absolute . . . 70
10.3 ORG . . . 71
10.4 Reserving space: DS . . . 71
10.5 Named segments . . . 72

10.6 Transient data segments . . . 73
10.7 Messages . . . 75

11 Data 77
11.1 DB: bytes . . . 77
11.2 Strings . . . 78
11.3 DW: words . . . 78
11.4 DC: strings that mark their own end . . . 79
11.5 The other names . . . 79
11.6 Messages . . . 80

12 Defining and sharing symbols 81
12.1 Three ways to give a name a value . . . 81
12.2 Labels . . . 82
12.3 EQU . . . 82
12.4 DEFL . . . 83
12.5 Sharing names between modules . . . 83
12.6 END and the start address . . . 85
12.7 Messages . . . 86

13 Including files 87
13.1 INCLUDE . . . 87
13.2 Where TATARA looks . . . 89
13.3 The TATARA environment variable . . . 90
13.4 Paths in the filename . . . 90
13.5 Files that include files . . . 90
13.6 Including a file twice . . . 91
13.7 END in an include file . . . 92
13.8 Messages . . . 92

14 The include files 95
14.1 What the files hold . . . 95
14.2 Including them . . . 96
14.3 MSXDOS.INC and the system macro . . . 97
14.4 BIOS.INC and SUBROM.INC . . . 98
14.5 ERRORS.INC . . . 99
14.6 The other files . . . 100

15 Conditional assembly 103
15.1 IF, ELSE and ENDIF . . . 103
15.2 The two passes . . . 105
15.3 IFDEF and IFNDEF . . . 106
15.4 IF1 and IF2 . . . 107
15.5 IFB, IFNB, IFIDN and IFDIF . . . 107
15.6 Nesting . . . 108
15.7 Messages . . . 108

16 Macros 111
16.1 Defining and using a macro . . . 112

16.2 Arguments . . . 113
16.3 Joining text with & . . . 114
16.4 Comments in a body . . . 115
16.5 LOCAL . . . 115
16.6 EXITM . . . 116
16.7 Macros inside macros . . . 117
16.8 Upper and lower case . . . 118
16.9 Messages . . . 118

17 Repeat blocks 121
17.1 REPT . . . 121
17.2 IRP . . . 122
17.3 IRPC . . . 124
17.4 LOCAL, EXITM and nesting . . . 124
17.5 Messages . . . 125

18 The listing 127
18.1 A listing line . . . 127
18.2 Pages . . . 129
18.3 Turning the listing off and on . . . 132
18.4 Macro expansions . . . 132
18.5 Lines not assembled . . . 134
18.6 Messages . . . 135

19 The symbol table dump and the diagnostic switches 137
19.1 The symbol table . . . 137
19.2 Reading the table . . . 139
19.3 The table and the listing . . . 142
19.4 Diagnostic switches . . . 142

20 Error messages 147
20.1 What a message says . . . 147
20.2 TATARA stops at the first error . . . 149
20.3 The trail . . . 150
20.4 Errors found on the second pass . . . 153
20.5 Internal errors . . . 153

III The linker, Tanren 155

21 Running TANREN 157
21.1 The command line . . . 157
21.2 The output file . . . 160
21.3 The options . . . 160
21.4 What TANREN prints . . . 161
21.5 The 127-character command line . . . 163
21.6 When TANREN does not start . . . 163

22 Link files **165**
22.1 A first link file . . . 165
22.2 Writing a link file . . . 166
22.3 Link files and the command line . . . 167
22.4 Messages . . . 168

23 Where TANREN looks for object files **169**
23.1 Three places . . . 169
23.2 The TANREN environment variable . . . 171
23.3 Paths in the name . . . 172
23.4 Where the output file goes . . . 172
23.5 Messages . . . 172

24 How a program is laid out **175**
24.1 Code, then data . . . 175
24.2 Named segments . . . 177
24.3 Moving the code and the data: /P: and /D: . . . 178
24.4 Transient segments across modules . . . 179
24.5 Absolute segments . . . 180
24.6 Messages . . . 181

25 Symbols across modules **183**
25.1 How names are matched . . . 183
25.2 What an external's value can be . . . 185
25.3 Names that no module defines . . . 186
25.4 Names defined twice . . . 187
25.5 Capitals and small letters . . . 188
25.6 Every module is linked . . . 189
25.7 /M in full . . . 190
25.8 Messages . . . 191

26 Output formats **193**
26.1 What TANREN writes . . . 193
26.2 A .COM program . . . 194
26.3 A raw image for another address . . . 194
26.4 A file for BLOAD: /B . . . 195
26.5 The entry address . . . 196
26.6 A ROM image . . . 198
26.7 How large a program can be . . . 199
26.8 When there is nothing to write . . . 200
26.9 Messages . . . 200

27 Linker error messages **203**
27.1 What a message says . . . 203
27.2 TANREN stops at the first error . . . 204
27.3 Damaged object files . . . 205
27.4 Segments . . . 206
27.5 Messages . . . 207

IV Worked guides **209**

28 A program in several modules **211**
28.1 The example . . . 211
28.2 The two modules . . . 212
28.3 The link file . . . 213
28.4 Building and running it . . . 214
28.5 What TANREN did . . . 215
28.6 Things to try . . . 216

29 Scratch variables shared through a transient segment **219**
29.1 The example . . . 219
29.2 The source . . . 220
29.3 Building it . . . 221
29.4 Reading the symbol table . . . 222
29.5 Reading the maps . . . 223
29.6 Things to try . . . 224

30 Macros and the listing modes **229**
30.1 The example . . . 229
30.2 The source . . . 230
30.3 Building and running it . . . 231
30.4 The three listing modes . . . 233
30.5 Things to try . . . 236

31 Long symbol names and case-sensitive names **239**
31.1 The example . . . 239
31.2 The source . . . 239
31.3 Building it . . . 241
31.4 Reading the two files . . . 241
31.5 Long symbol names across modules . . . 243
31.6 Macro names and /C . . . 244

32 A binary for MSX-BASIC's BLOAD **247**
32.1 The example . . . 247
32.2 The source . . . 247
32.3 Building it . . . 249
32.4 Running it from BASIC . . . 250
32.5 Where the file can go . . . 250
32.6 Things to try . . . 252

33 A 16 KB ROM cartridge **255**
33.1 The example . . . 255
33.2 The source . . . 255
33.3 Building it . . . 258
33.4 Running it . . . 259
33.5 Things to try . . . 259

34 A project in several directories **263**
34.1 The example . . . 263
34.2 The sources . . . 264
34.3 Building and running it . . . 266
34.4 Things to try . . . 267

35 Building TATARA and TANREN from their own sources **271**
35.1 The sources . . . 271
35.2 BUILD.BAT . . . 272
35.3 Running it . . . 275
35.4 Building the tools with themselves . . . 276

A Directive reference **279**
A.1 The directives by purpose . . . 279
A.2 The directives in alphabetical order . . . 282
A.3 Labels on directive lines . . . 288
A.4 Other names . . . 289

B Instruction summary **291**
B.1 How to read the tables . . . 292
B.2 8-bit loads . . . 293
B.3 16-bit loads, PUSH and POP . . . 294
B.4 Exchange, block transfer and search . . . 295
B.5 8-bit arithmetic and logic . . . 296
B.6 General purpose and CPU control . . . 297
B.7 16-bit arithmetic . . . 298
B.8 Rotate and shift . . . 299
B.9 Bit set, reset and test . . . 301
B.10 Jumps . . . 301
B.11 Calls, returns and restarts . . . 302
B.12 Input and output . . . 303
B.13 The undocumented instructions . . . 304
B.14 The R800 multiplications . . . 305
B.15 Forms that are not accepted . . . 305

C Operators and precedence **309**
C.1 The operators . . . 309
C.2 At the edges . . . 311
C.3 Values . . . 311

D Error messages **313**
D.1 How a message is laid out . . . 313
D.2 TATARA's messages . . . 314
D.3 TANREN's messages . . . 318
D.4 TANREN's warnings and notes . . . 320

E Limits **321**
E.1 TATARA . . . 321
E.2 TANREN . . . 322

E.3 The command line . . . 323
E.4 What has no fixed limit . . . 323

F The `.tro` object file format 325
F.1 What an object file says . . . 325
F.2 A worked example . . . 326
F.3 File layout . . . 329
F.4 The records . . . 330
F.5 How TANREN reads an object file . . . 332

G Memory 335
G.1 The memory map . . . 335
G.2 What goes into the mapper . . . 336
G.3 How much memory the tools need to build a program . . . 336
G.4 When the mapper is full . . . 338

H Where to get the hardware 339
H.1 What the tools need . . . 339
H.2 The cartridges . . . 340
H.3 The makers and where to buy . . . 340
H.4 Availability . . . 341

I Environment variables 343
I.1 Setting a variable . . . 343
I.2 PATH . . . 344
I.3 TATARA . . . 344
I.4 TANREN . . . 345
I.5 The rules both share . . . 346

J Source compatibility with M80 349
J.1 What carries over . . . 349
J.2 M80 directives that TATARA does not accept . . . 350
J.3 Other forms . . . 351
J.4 What TATARA adds . . . 352
J.5 Other differences . . . 353
J.6 Converting a source . . . 353

K The include files 355
K.1 MSXDOS.INC . . . 356
K.2 ERRORS.INC . . . 356
K.3 BIOS.INC . . . 357
K.4 WORKAREA.INC . . . 358
K.5 SUBROM.INC . . . 358
K.6 HOOKS.INC . . . 358
K.7 PORTS.INC . . . 358
K.8 ASCII.INC . . . 359
K.9 EXTBIO.INC . . . 359
K.10 Further reading . . . 359

L Glossary **361**

About this manual

Who it is for

This manual is for MSX users who want to write their own programs in Z80 assembly language, and in particular for those who are just starting. It assumes that you know how to use MSX-DOS: how to change drives and directories, list files and run programs. It does not assume that you have used an assembler or a linker before.

If you already program in assembly, you can skip chapter 2. The reference material you are most likely to want is in Parts II and III and in the appendices.

What it does not teach

This manual explains how to use Tatara and Tanren. It does not teach the Z80 instruction set, or how to design and write a program with it; for that you will need a book or course on Z80 assembly programming.

How it is organised

The manual is divided into four parts and a set of appendices.

Part I, Getting started, says what the tools are, explains how an assembler and a linker work together, installs them, and takes you through building a first program.

Part II, The assembler, Tatara, describes the assembler: how to run it, how a source file is written, and every feature of the language, from expressions and instructions to macros and conditional assembly.

Part III, The linker, Tanren, describes the linker: how to run it, how it finds and joins object files, where it places the program in memory, and the kinds of file it can write.

Part IV, Worked guides, builds a complete program for each of a number of common tasks, such as a program in several files, a binary for MSX-BASIC, or a ROM cartridge.

The appendices are for looking things up: every directive, instruction, operator, error message and limit in one place.

If you are new to assembly, read Part I in order, and then work through the guides in Part IV, turning to Parts II and III when a guide uses something you want to understand in full. If you are an experienced programmer, chapters 5 and 21 and the appendices will give you most of what you need.

Conventions

What you type is shown in bold typewriter type, as in **`tatara hello.as hello.tro`**. What a program prints is shown in plain typewriter type, as in `ERROR: undefined symbol in an expression.`

An example of a session at the MSX-DOS2 prompt shows both. The prompt is shown as `A:\>`. When the current directory is not the root directory of the drive, the prompt includes it, as in `A:\SOMEDIR\>`:

MSX-DOS

```
A:\>tanren /v
Tatara MSX Linker v1.2.0
Copyright (C) 2026 Javier Lavandeira
https://tatara.tools
```

File names are written in upper case in the text, such as `HELLO.AS`, because that is how MSX-DOS and both programs show them. You can type them in upper or lower case.

Hexadecimal numbers are written the way the assembler reads them, with a trailing `h`: `0100h` is the hexadecimal number 100, which is 256 in decimal.

Tatara and *Tanren* are the names of the two tools. When the manual means the programs themselves, the files you run, it writes their names in capitals: TATARA and TANREN.

> Note. A note gives information that is useful but that you can skip on a first reading.

> Warning. A warning describes something that can lose your work or produce a program that does not do what you expect.

Which version it describes

This manual describes version 1.2.0 of Tatara and Tanren. The latest version of both tools, and of this manual, is published at `https://tatara.tools`. The programs, and the example programs used in this manual, can be downloaded from `https://tatara.tools/download`.

PART I

Getting started

1 Introduction 3

2 How an assembler and a linker work 7

3 Installation 17

4 A first program 23

CHAPTER 1

Introduction

1.1 What Tatara and Tanren are 3
1.2 Where the names come from 4
1.3 What they can do 4
1.4 What you need 6
1.5 What is next 6

1.1 What Tatara and Tanren are

Tatara and Tanren are the two tools you need to turn a program written in Z80 assembly language into a program that runs on an MSX.

Tatara is an *assembler*. It reads the program you have written, a text file of instructions such as `ld a,5` and `call 5`, and turns it into the machine code that the Z80 processor understands.

Tanren is a *linker*. It takes the machine code that Tatara has produced, from one file or from several, joins it together, decides where in memory each part will go, and writes a finished program that you can run.

Why there are two tools and not one, and what each of them does in detail, is the subject of chapter 2.

Both tools run on the MSX itself, under MSX-DOS2 or Nextor. You write your program, assemble it, link it and run it on the same machine, with no other computer involved. They run in the same way in an MSX emulator.

Tatara reads the assembly language of Microsoft's MACRO-80, the assembler that was used to write much of the software of the CP/M and early MSX years. Programs and books written for MACRO-80 can therefore be used with Tatara. Appendix J lists where the two differ.

1.2 Where the names come from

The two names come from the traditional way of making a Japanese sword.

A *tatara* (たたら) is a clay furnace. Iron sand and charcoal are loaded into it and burnt, over about three days, until they have turned into raw steel: a rough block of metal, strong but shapeless, and not yet a blade.

Tanren (鍛錬) is the forging that follows. The smith heats the raw steel, hammers it and folds it, again and again, until it becomes a finished blade. In everyday Japanese the same word also means training and discipline.

The two tools do the same two jobs for a program, as figure 1.1 shows. The source files you write are the raw materials. TATARA turns them into object files: real machine code, but not yet a program you can run, in the same way that raw steel is not yet a sword. TANREN takes one or more object files and forges them into the finished program. Chapter 2 builds on this picture.

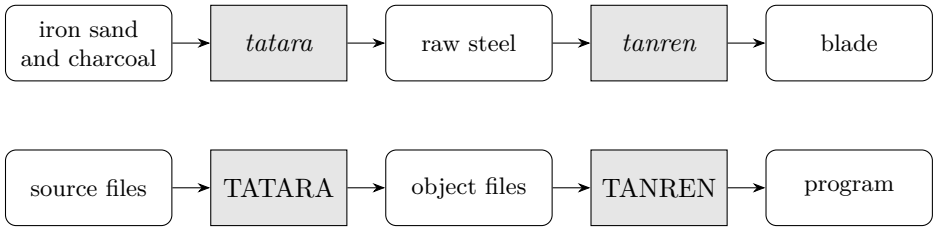


Figure 1.1: From raw materials to a finished blade, and from source files to a finished program.

1.3 What they can do

This is a short list, for a first reading. Each point is explained in full in the chapter it refers to.

- **Long symbol names.** The name of a symbol, such as a label, can be up to 255 characters long, so it can say what it is for. Names can also be made case-sensitive with the `/C` option, so that `Count` and `count` are different names (chapter 7).
- **Macros and conditional assembly.** A piece of code can be written once and used many times, and parts of a program can be included or left out when it is assembled (chapters 15 to 17).

- **Segments.** Code and data can be kept in separate areas of the program, called segments, which can be given names. Data areas that are only needed by one part of a program at a time can share the same memory (chapter 10).

- **Programs in many files and directories.** A file that is included with the `INCLUDE` directive is looked for in three places, and the `TATARA` and `TANREN` environment variables say where else to look (chapters 13 and 23).

- **Ready-made names.** Tatara comes with include files that name the MSX-DOS2 functions and error codes, the BIOS entry points, the system work area and much more, 904 names in all, so that a program can write `_STROUT` instead of looking up that it is `09h` (chapter 14).

- **Large programs.** Both tools keep their tables, and Tanren keeps the program it is building, in the memory mapper. The size of the program you can build is limited by how much memory your MSX has, not by the memory MSX-DOS leaves free (chapter 26 and appendix G).

- **Instructions for the linker in a file.** MSX-DOS cuts off a command line at 127 characters, which a program built from many files soon reaches. Tanren can read the list of files, and everything else it would be told on the command line, from a text file instead (chapter 22).

- **Errors that point to your line.** When there is an error, the message names the file and the line you wrote, even when the line came from a macro or from an included file (chapter 20).

- **Several kinds of output.** Tanren can write a program for MSX-DOS, a binary for MSX-BASIC's `BLOAD`, an image of the code for any address, or the image of a ROM cartridge (chapter 26).

- **R800 instructions.** Tatara also assembles `MULUB` and `MULUW`, the multiplication instructions of the R800 processor in the MSX turbo R (chapter 9).

- **Undocumented instructions.** Tatara assembles the undocumented Z80 instructions that MSX programs use, such as `SLL` and those that use `IXH` and `IXL` (section 9.5).

1.4 What you need

To use Tatara and Tanren you need:

- An MSX running **MSX-DOS2** or **Nextor**. Under MSX-DOS1 both programs stop at once with a message saying so.
- A **memory mapper**. MSX-DOS2 cannot run without one, so if your MSX runs MSX-DOS2 it already has one. How much memory the tools need to build a program of a given size is described in appendix G.
- Tatara's archive, `TATARxyz.ZIP`, from `https://tatara.tools/download`, and `UNKAGO.COM`, from `https://kago.tools/en/download/`, to unpack it. Chapter 3 explains how to install them.
- An editor that saves plain text files, to write your programs with.

Any MSX2 or later model that can run MSX-DOS2 or Nextor will do, and so will any MSX emulator that can run them.

1.5 What is next

Chapter 2 explains what an assembler and a linker do, and why a program is built in two steps. Chapter 3 installs the tools, and chapter 4 uses them to build and run a first program.

CHAPTER 2

How an assembler and a linker work

2.1 Two steps 7
2.2 What the assembler does 8
2.3 What the assembler cannot know 9
2.4 What the linker does 10
2.5 A worked example 10
2.6 Code and data: segments 13
2.7 Why two steps are worth it 14
2.8 The words in this chapter 14

This chapter explains what TATARA and TANREN each do, and why a program is built in two steps rather than one. It introduces the words that the rest of the manual uses, and follows one small program from the source files you write to the program you run.

2.1 Two steps

A program starts as one or more *source files*: text files that you write with an editor, in assembly language. Tatara's source files are normally named with the extension `.AS`.

TATARA reads a source file and writes an *object file*, with the extension `.TRO`. An object file holds the program's machine code, but it is not yet a program that can be run.

TANREN reads one or more object files and writes the finished *program file*. For

MSX-DOS2 this is a file with the extension `.COM`, which you run by typing its name at the MSX-DOS2 prompt. Figure 2.1 shows the two steps.

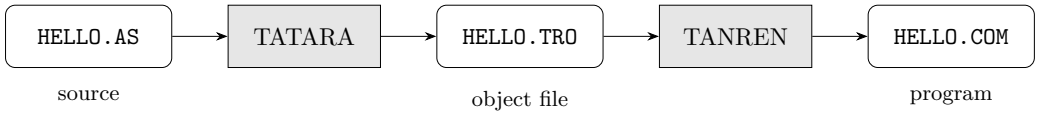


Figure 2.1: The two steps from source to program.

In the terms of chapter 1, the source files are the raw materials, the object files are the raw steel, and the program is the finished blade. The rest of this chapter explains what happens in each step, and why the raw steel is worth having.

2.2 What the assembler does

A source file is a list of lines. Most of them are instructions for the Z80, written as short words that a person can read, such as `ld c,_STROUT`. The Z80 cannot read words. What it reads are bytes: numbers from 0 to 255, stored one after another in memory, which together are the program's *machine code*. Turning each line of a source file into its bytes is called *assembling*.

Table 2.1 shows three lines and the bytes TATARA assembles them into. The bytes are written in hexadecimal.

Line	Bytes
`ld c,_STROUT`	`0E 09`
`call BDOS`	`CD 05 00`
`ret`	`C9`

Table 2.1: Three lines and the bytes they become.

The first byte of each line says which instruction it is. The bytes after it are the instruction's operand, if it has one. An address takes two bytes, and the Z80 stores the low byte first, so the address `0005h` is stored as `05` followed by `00`.

`_STROUT` and `BDOS` are not instructions. They are names for the numbers `09h` and `0005h`, and they are defined in `MSXDOS.INC`, one of the include files that come with Tatara (chapter 14). A program that includes that file can use the names, and the assembler replaces each name with its number. A name that stands for a number in this way is called a *symbol*.

A program also needs names for places in itself: the start of a loop, or a message to be printed. A *label* is such a name. It is written at the start of a line, followed by a colon, as in `msg1:`, and it stands for the *address* of the first byte that line produces: the number of the memory location where that byte will be. A label is a symbol too.

To know what each label stands for, the assembler counts the bytes as it goes. If a program's first line produces three bytes and its second line produces three more, the third line starts six bytes after the first.

> Note. This manual assumes that you know what a byte, a hexadecimal number and a memory address are. The book or course you learn Z80 programming from explains them.

2.3 What the assembler cannot know

Counting bytes tells the assembler how far each label is from the start of the program. There are two things that a single source file cannot tell it.

1. **Where the program will be in memory.** The assembler knows that a label is, say, 17 bytes after the start of the code, but it does not know at which address the code will start. That is decided later, when all the parts of the program are put together. An address that depends on where the code is placed is *relocatable*.
2. **Where things in other files are.** A program can be split over several source files, and a line in one of them can call a routine that is written in another. TATARA assembles one file at a time, so it cannot know the address of a name that is defined in a different file.

TATARA therefore does all the work it can, and writes down what it could not finish. Where an address is still unknown, it leaves a *hole* in the machine code, and it records what belongs in the hole. The result is the object file: machine code with holes in it, and a note for each hole. This is the raw steel.

The code of one source file, as it is in its object file, is called a *module*. A module can offer some of its symbols to other modules, and it can use symbols that other modules offer:

- A *public symbol* is one that a module defines and makes available to other modules. It is declared with the `PUBLIC` directive.

- An *external symbol* is one that a module uses but that another module defines. It is declared with the `EXTRN` directive.

A directive is a line that gives an instruction to the assembler rather than to the Z80; it produces no machine code of its own.

2.4 What the linker does

TANREN reads one or more object files and does three things.

1. **It places the modules** one after another in memory, starting at address `0100h`, which is where MSX-DOS2 loads a program and starts it.[1]
2. **It fills every hole.** For a relocatable address, it adds the address where the module was placed to the value that TATARA left in the hole. For an external symbol, it finds the module that declared the symbol public and writes the symbol's address into the hole. Filling the holes is called *relocation*.
3. **It writes the program file**: the finished bytes, in a file that MSX-DOS2 can load and run.[2]

The whole job is called *linking*. It is the forging that turns the raw steel into a blade.

2.5 A worked example

The example `TWOMOD` is a program in two modules. It is one of the examples that come with Tatara; chapter 4 builds it, and chapter 28 returns to it. The first module, `MAIN.AS`, prints two lines by calling a routine named `putstr`, which it declares external:

[1] This is the default, because TANREN builds MSX-DOS2 programs unless it is told otherwise. It can be given another start address, to build a program that runs somewhere else in memory, such as a ROM cartridge or a binary for MSX-BASIC's `BLOAD` (chapter 24).

[2] Again, this is the default. TANREN can also write a binary for MSX-BASIC's `BLOAD`, or the image of a ROM cartridge (chapter 26).

```
; MAIN.AS - one of two modules.
;
; It knows that putstr exists. It does not know where putstr is, and
; it does not need to: EXTRN says the name is somebody else's, and the
; linker fills the address in.

                extrn   putstr          ; defined in PUTSTR.AS

                include msxdos.inc      ; BDOS and _TERM0

                cseg

start:          ld      hl,msg1
                call    putstr
                ld      hl,msg2
                call    putstr
                ld      c,_TERM0
                jp      BDOS

msg1:           db      "Two modules, one program.",13,10,0
msg2:           db      "The linker joined them.",13,10,0

                end     start
```

The second module, `PUTSTR.AS`, defines `putstr` and declares it public:

```
; PUTSTR.AS - the other module.
;
; PUBLIC is what makes the name visible outside this file. Without it
; the name would still assemble here and the link of MAIN.AS would
; stop with an undefined symbol - which is worth trying once.

                public  putstr

                include msxdos.inc      ; BDOS, _CONOUT and "system"

                cseg

; putstr - print a string.
;
;   HL is pushed around the BDOS call because MSX-DOS promises nothing
;   about the registers it hands back.
;
; Input:        HL -> the string, ending in a zero byte
; Output:       it is printed
; Modifies:     AF, BC, DE, HL

putstr:         ld      a,(hl)
                or      a
                ret     z
                push    hl
                ld      e,a
```

```
                system  _CONOUT
                pop     hl
                inc     hl
                jr      putstr

                end
```

Both modules include `MSXDOS.INC`. The `INCLUDE` line reads that file as if its lines were written at that point, and it produces no bytes of its own. `PUTSTR.AS` uses `system _CONOUT`: `system` is a *macro* defined in `MSXDOS.INC`, a single name that stands for a group of lines. Here it stands for `ld c,_CONOUT` followed by `call BDOS`. Chapter 16 explains macros.

2.5.1 What TATARA writes for MAIN.AS

`MAIN.AS` assembles to 71 bytes of machine code with four holes in it, as table 2.2 shows. The offset is the number of bytes from the start of the module.

Offset	Line	Bytes	Hole
0	`ld hl,msg1`	`21 11 00`	relocatable: `msg1`, 17 bytes in
3	`call putstr`	`CD 00 00`	external: `putstr`
6	`ld hl,msg2`	`21 2D 00`	relocatable: `msg2`, 45 bytes in
9	`call putstr`	`CD 00 00`	external: `putstr`
12	`ld c,_TERM0`	`0E 00`	none
14	`jp BDOS`	`C3 05 00`	none: `BDOS` is always `0005h`
17	`msg1: db ...`	28 bytes of text	none
45	`msg2: db ...`	26 bytes of text	none

Table 2.2: The module `MAIN`, as TATARA writes it.

A relocatable hole holds the label's offset: `11h` is 17, the offset of `msg1`. An external hole holds zero, because the assembler knows nothing about `putstr` except its name. The `jp BDOS` at offset 14 is not a hole: `BDOS` is the fixed address `0005h`, the same wherever the program is placed.

2.5.2 What TATARA writes for PUTSTR.AS

`PUTSTR.AS` assembles to 14 bytes, and it has no holes at all. The `call BDOS` inside `system _CONOUT` calls a fixed address. The `jr putstr` at its end does not hold

an address either: it holds a distance, "jump back 14 bytes", which is the same wherever the code is placed. The module has one public symbol, `putstr`, at offset 0.

2.5.3 What TANREN does

TANREN places `MAIN` first, because it is named first, and `PUTSTR` straight after it. Then it fills the four holes, as table 2.3 shows.

`MAIN` placed at	`0100h` (71 bytes, to `0146h`)
`PUTSTR` placed at	`0147h` (14 bytes, to `0154h`)
so `putstr` is	`0147h`
hole at `0101h`	`0011h` + `0100h` = `0111h`, the address of `msg1`
hole at `0104h`	`0147h`, the address of `putstr`
hole at `0107h`	`002Dh` + `0100h` = `012Dh`, the address of `msg2`
hole at `010Ah`	`0147h`, the address of `putstr`

Table 2.3: How TANREN links `MAIN` and `PUTSTR`.

The result is `TWOMOD.COM`, 85 bytes long, which MSX-DOS2 loads at `0100h`. Figure 2.2 shows the two object files and the program they become.

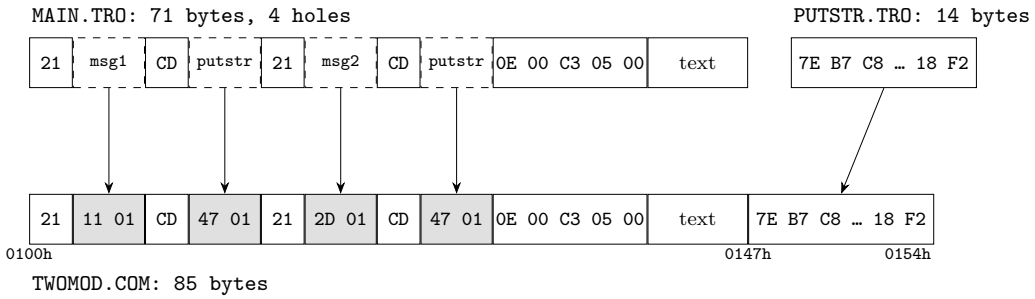


Figure 2.2: Linking `MAIN.TRO` and `PUTSTR.TRO` into `TWOMOD.COM`.

2.6 Code and data: segments

A program holds two kinds of thing: instructions, and the data they work on. Tatara lets you keep them apart, in two *segments*. The *code segment* is started with the `CSEG` directive, and the *data segment* with `DSEG`.

When TANREN places the modules, it puts the code segments of all the modules first, one after another, and the data segments after them. In the finished program each kind is therefore in one piece.

`TWOMOD` keeps its two messages in the code segment, which is fine for a program this small, and its data segment is empty. Chapter 10 explains segments in full, including segments with names of their own and data areas that different parts of a program can share.

2.7 Why two steps are worth it

Building a program in two steps takes one more command than building it in one, and it gives four things in return.

Reuse. A module such as `PUTSTR` can be linked into any number of programs without being assembled again.

Size. A large program is easier to work on when it is split over several files, and only the file that changed needs to be assembled again.

Collaboration. A programmer can write a library of routines, assemble it, and make its object files available. Other programmers link it into their own programs without copying its code into theirs, and without needing its source files at all: the object file is enough.

Choice of output. Where the program is placed is decided only at the end, so the same object files can become a program for MSX-DOS2, a binary for MSX-BASIC's `BLOAD`, or a ROM cartridge (chapter 26).

2.8 The words in this chapter

Table 2.4 lists the terms this chapter has introduced. The glossary (appendix L) lists them with every other term in the manual.

Term	Meaning
source file	the text file you write, in assembly language
machine code	the bytes the Z80 reads and runs
assembling	turning source lines into machine code
symbol	a name that stands for a number
label	a symbol that stands for an address in the program
address	the number of a memory location
relocatable	an address that depends on where the code is placed
hole	a place in the machine code where an address is still unknown
object file	machine code with holes, written by TATARA
module	the code of one source file, in its object file
public symbol	a symbol a module offers to other modules
external symbol	a symbol a module uses but another module defines
relocation	filling the holes
linking	placing the modules, filling the holes, and writing the program
program file	the finished program, written by TANREN
segment	an area of the program for one kind of content: code or data

Table 2.4: The words introduced in this chapter.

CHAPTER 3

Installation

3.1 What you get 17
3.2 Where the files go 18
3.3 Installing, step by step 19
3.4 Setting PATH and TATARA 20
3.5 Checking the installation 21
3.6 Updating 22

This chapter takes you from the downloaded files to an MSX on which TATARA and TANREN run from any directory and find the include files. The installation it describes is the one the rest of the manual assumes: every session from here on is shown with these directories.

3.1 What you get

Tatara is distributed as one *archive*, a file that holds several files packed together. It is named `TATARxyz.ZIP`, where `xyz` is the version: `TATAR120.ZIP` is version 1.2.0. You download it from `https://tatara.tools/download`. It holds a whole installation, already arranged in directories, as table 3.1 shows.

The archive is unpacked on the MSX with UNKAGO, a decompressor for ZIP, LZH and PMA archives. UNKAGO is required: it is what unpacks the archive on the MSX, and it does not come with MSX-DOS2 or Nextor. Download `UNKAGO.COM` from `https://kago.tools/en/download/`. Like TATARA and TANREN, it runs under MSX-DOS2 or Nextor.

File or directory	What it holds
`BIN`	`TATARA.COM`, the assembler, and `TANREN.COM`, the linker
`INCLUDE`	the include files (chapter 14)
`EXAMPLES`	the example programs of this manual, one directory each
`README.TXT`	the version, and the address of the web site
`LICENSE`	the Apache License 2.0, under which Tatara is distributed

Table 3.1: What the archive holds.

> Note. Earlier releases were distributed as `.LZH` archives, such as `TATAR117.LZH`. They hold no directories, so they are not installed as this chapter describes. This chapter describes the `.ZIP` archive.

This manual assumes that you download the two files on another computer, such as a PC or a Mac, and copy them to the MSX, for instance on a memory card.

3.2 Where the files go

The manual assumes that the archive is unpacked into `A:\TATARA`, which gives the directories below.

```
A:\TATARA\
    BIN\         TATARA.COM and TANREN.COM
    INCLUDE\     the include files
    EXAMPLES\    the example programs, one directory each
    README.TXT
    LICENSE
```

Any other place works just as well, as long as two things are true: the directory that holds the two programs is on the `PATH`, and the `TATARA` environment variable names the directory that holds the include files. Section 3.4 explains both. The rest of the manual shows `A:\TATARA`; if you install elsewhere, read your own directories in its place.

> Note. A Japanese MSX shows the backslash as a yen sign, so the directory above appears as `A:¥TATARA¥INCLUDE`. It is the same character, and the paths in this manual work as they are.

3.3 Installing, step by step

Copy `TATAR120.ZIP` and `UNKAGO.COM` to the MSX, both into the same directory. Any directory will do, because UNKAGO is told where to put the files; the example below has them in the root directory of drive A:.

1. **Unpack the archive into `A:\TATARA`.** In the directory that holds the two files, run UNKAGO with the switch `/D:` and the directory to unpack into. UNKAGO creates `A:\TATARA` if it does not exist, and the directories inside it.

   MSX-DOS

   ```
   A:\>unkago /d:a:\tatara tatar120.zip
   UNKAGO LZH/PMA/ZIP Decompressor v1.0.1
   Copyright (C) 2026 Javier Lavandeira
   https://kago.tools

   Extracting BIN\ OK
   Extracting BIN\TATARA.COM OK
   Extracting BIN\TANREN.COM OK
   Extracting INCLUDE\ OK
   Extracting INCLUDE\WORKAREA.INC OK
   ...
   Extracting EXAMPLES\TWOMOD\TWOMOD.LNK OK
   Extracting README.TXT OK
   Extracting LICENSE OK
   ```

   UNKAGO prints one line for each directory and file in the archive, fifty in all, and each line should end in `OK`. The list above is shortened in the middle.

2. **Check.** **`dir a:\tatara\bin`** should list `TATARA.COM` and `TANREN.COM`, **`dir a:\tatara\include`** the ten files in table 3.2, and **`dir a:\tatara\examples`** eight directories, one for each example.

3. **Tidy up.** `TATAR120.ZIP` is not needed any more, and you can delete it. Keep `UNKAGO.COM`: an update is unpacked with it too (section 3.6).

File	What it holds
`ASCII.INC`	the control codes
`BIOS.INC`	the MAIN ROM entry points
`ERRORS.INC`	the MSX-DOS and MSX-DOS2 error codes
`EXTBIO.INC`	the extended BIOS, and the MSX-MUSIC FM BIOS
`HOOKS.INC`	the hooks
`MSXDOS.INC`	the MSX-DOS function numbers, and the `system` macro
`PORTS.INC`	the I/O ports
`SUBROM.INC`	the SUB ROM entry points
`WORKAREA.INC`	the system work area
`README.TXT`	a short description of the files

Table 3.2: The files in `A:\TATARA\INCLUDE`.

3.4 Setting PATH and TATARA

An *environment variable* is a name that holds a piece of text, which MSX-DOS2 keeps in memory and which programs can read. The installation needs two of them.

`PATH` When you type a command, MSX-DOS2 looks for a program of that name in the current directory, and then in every directory named in `PATH`. Adding `A:\TATARA\BIN` to it lets you run TATARA and TANREN from any directory.

`TATARA` When a source file includes another file, TATARA looks for it in the directory of the source file, then in the current directory, and then in every directory named in `TATARA`. Setting it to `A:\TATARA\INCLUDE` lets every program include the files of table 3.2 without a copy of them. Chapter 13 gives the full rule.

To set them for the session you are in, type:

MSX-DOS

```
A:\>set path=%path%;a:\tatara\bin
A:\>set tatara="a:\tatara\include"
```

`%path%` stands for what `PATH` already holds, so the first line adds the new directory to the end of it rather than replacing it. Type **`set`** on its own to see every environment variable and its value; your `PATH` will show directories of your own before the new one.

> Note. At the MSX-DOS2 prompt, write the value of `TATARA` between quotation marks, as above. Without them, MSX-DOS2 answers `*** Invalid environment string` and does not set the environment variable. In `AUTOEXEC.BAT`, below, the quotation marks are not needed.

An environment variable set this way lasts until the MSX is reset or switched off. To set both every time the MSX starts, add the two lines below to `AUTOEXEC.BAT`, the batch file in the root directory of the boot drive that MSX-DOS2 runs when it starts. If you do not have one, create it with a text editor. It can already hold commands of its own; add the two lines at the end:

```
set path=%path%;a:\tatara\bin
set tatara=a:\tatara\include
```

> Note. `TATARA` can name several directories, separated by semicolons, in the same way as `PATH`. This is useful when a project has include files of its own (chapter 13).

> Note. TANREN has an environment variable of its own, `TANREN`, which names the directories where it looks for object files. This installation does not need it (chapter 23).

3.5 Checking the installation

From any directory, run each program with the option `/V`. It prints the program's name and version, and does nothing else.

MSX-DOS

```
A:\>tatara /v
Tatara MSX Macro-Assembler v1.2.0
Copyright (C) 2026 Javier Lavandeira
https://tatara.tools


A:\>tanren /v
Tatara MSX Linker v1.2.0
Copyright (C) 2026 Javier Lavandeira
https://tatara.tools
```

If MSX-DOS2 answers as below instead, `PATH` does not include `A:\TATARA\BIN`. Check it with **`set`**, and set it again as in section 3.4.

MSX-DOS

```
A:\>tatara /v
*** Unrecognized command
```

The `TATARA` environment variable is checked in chapter 4, when the first program includes `MSXDOS.INC`.

3.6 Updating

A new version is unpacked in the same way, with the switch `/O` added:

MSX-DOS

```
A:\>unkago /o /d:a:\tatara tatarxyz.zip
```

Without `/O`, UNKAGO does not replace files that already exist: it prints `Skipping BIN\TATARA.COM: it already exists` for each one, and the old version stays. `/V` shows which version each program is.

> Warning. `/O` replaces the examples too. Before you change an example, copy its directory somewhere else and work on the copy, or the next update will overwrite your changes.

CHAPTER 4

A first program

4.1 Before you start 23
4.2 The program 24
4.3 Assembling it 25
4.4 Linking it 26
4.5 Running it 26
4.6 The build file 27
4.7 A program in two modules 27
4.8 What is next 29

This chapter builds and runs two of the examples: `HELLO`, a program in one source file, and `TWOMOD`, the program in two modules from chapter 2. It shows every command, and what TATARA and TANREN print in reply.

4.1 Before you start

You need the installation of chapter 3: the two programs on the `PATH`, the `TATARA` environment variable set, and the examples in `A:\TATARA\EXAMPLES`. This chapter works in `A:\TATARA\EXAMPLES\HELLO`, and then in `A:\TATARA\EXAMPLES\TWOMOD`. Go to the first of them:

MSX-DOS

```
A:\>cd \tatara\examples\hello
A:\TATARA\EXAMPLES\HELLO>
```

4.2 The program

HELLO.AS is the smallest complete program: it prints a line and returns to MSX-DOS2.

```
; HELLO.AS - the smallest complete program.
;
; One source, one object, one .COM. It prints a line and gives the
; machine back to MSX-DOS.

                include msxdos.inc      ; BDOS, _STROUT, _TERM0 and the
                                        ;   "system" macro, found through
                                        ;   TATARA - see BUILD.BAT

                cseg                    ; RELOCATABLE code. Nothing here
                                        ;   knows its own address, and the
                                        ;   linker chooses one - 0100h for a
                                        ;   .COM, unless /P: says otherwise

start:          ld      de,msg
                system  _STROUT
                ld      c,_TERM0
                jp      BDOS

msg:            db      "Hello from Tatara.",13,10,"$"

                end     start           ; where it starts, for the linker
```

Taking it a few lines at a time:

- The first lines start with a semicolon. A semicolon starts a *comment*: TATARA ignores it and the rest of the line. Comments are for the people who read the program. Chapter 6 describes how a line is written.
- `include msxdos.inc` reads the file MSXDOS.INC at that point, as if its lines were written there. That file defines the names BDOS, _STROUT and _TERM0, and the macro `system` (chapters 13 and 14). TATARA finds it through the TATARA environment variable.
- `cseg` starts the code segment: what follows is relocatable code, and TANREN decides where it goes (chapter 10).
- `start:` and `msg:` are labels. `start` is the address of the first instruction, and `msg` the address of the text.
- `ld de,msg` and `system _STROUT` ask MSX-DOS2 to print the text at `msg`, up to the $ that ends it.

- `ld c,_TERM0` and `jp BDOS` end the program, and MSX-DOS2 shows its prompt again.
- `db` puts bytes into the program: the text, then 13 and 10, which move to the start of the next line, then the `$` (chapter 11).
- `end start` ends the source file, and tells TANREN that the program starts at `start`.

4.3 Assembling it

Assemble `HELLO.AS` into the object file `HELLO.TRO`:

MSX-DOS

```
A:\TATARA\EXAMPLES\HELLO>tatara hello.as hello.tro
Tatara MSX Macro-Assembler v1.2.0
Copyright (C) 2026 Javier Lavandeira
https://tatara.tools

ended at HELLO.AS(22)
```

TATARA prints its banner, reads the source file, and prints a summary: where the source ended, here at line 22 of `HELLO.AS`, the line with `END`. It has written `HELLO.TRO`, the object file of chapter 2: machine code with holes in it.

Warning. If `TATARA` is not set, TATARA cannot find `MSXDOS.INC`, and stops:

MSX-DOS

```
A:\TATARA\EXAMPLES\HELLO>tatara hello.as hello.tro
Tatara MSX Macro-Assembler v1.2.0
Copyright (C) 2026 Javier Lavandeira
https://tatara.tools

ERROR: cannot open MSXDOS.INC
    included from HELLO.AS(6)
```

The second line says where the file was asked for: line 6 of `HELLO.AS`. Set the environment variable as in section 3.4, and assemble again.

4.4 Linking it

Link `HELLO.TRO` into the program `HELLO.COM`:

MSX-DOS

```
A:\TATARA\EXAMPLES\HELLO>tanren /o:hello.com hello.tro
Tatara MSX Linker v1.2.0
Copyright (C) 2026 Javier Lavandeira
https://tatara.tools

1 module, 7 records, ends at EOF.
Wrote HELLO.COM, 0100-0121 (34 bytes), entry 0100.
```

The option `/O:` names the program file. Without it, TANREN names the program after the first object file, so here the name would be `HELLO.COM` anyway; this manual gives `/O:` so that the name can be seen in the command.

The first line of the summary says what TANREN read: one module, made of seven records, up to the end of the file (chapter 21). The second says what it wrote: `HELLO.COM`, which occupies the addresses `0100h` to `0121h`, 34 bytes, and starts at `0100h`. The directory now holds all three files:

MSX-DOS

```
A:\TATARA\EXAMPLES\HELLO>dir hello.*
 Volume in drive A: is MAIN
 Directory of A:\TATARA\EXAMPLES\HELLO

HELLO    AS        603 26-09-28 17:32
HELLO    TRO        93 26-09-29  1:27
HELLO    COM        34 26-09-29  1:27
 730 bytes in 3 files   12664K free
```

The name of the volume, the dates and the free space will be different on your MSX.

4.5 Running it

Type the program's name:

MSX-DOS

```
A:\TATARA\EXAMPLES\HELLO>hello
Hello from Tatara.
```

MSX-DOS2 loaded `HELLO.COM` at `0100h`, the address TANREN placed it at, and started it there. Two commands took the program from source file to running program.

4.6 The build file

Every example has a *build file*, `BUILD.BAT`: a batch file that runs the commands that build the example. `HELLO`'s runs the two commands of this chapter, with the option `/Q`, which leaves out the banner and the summary. An error is printed with or without `/Q`, so a run that prints nothing is a run that succeeded. Type **`type build.bat`** to read it.

> Warning. A build file sets the `TATARA` environment variable when it starts, and clears it when it finishes. After you run one, `TATARA` is empty for the rest of the session, and the next program that includes a file stops with `cannot open`. Set the environment variable again, as in section 3.4, or reset the MSX so that `AUTOEXEC.BAT` sets it.

This manual has you type the commands instead. It is the better way to learn them.

4.7 A program in two modules

`TWOMOD` is the program of chapter 2: `MAIN.AS` prints two lines by calling `putstr`, which is defined in `PUTSTR.AS`. Go to its directory, and assemble the two source files, one command for each:

MSX-DOS

```
A:\TATARA\EXAMPLES\HELLO>cd ..\twomod
A:\TATARA\EXAMPLES\TWOMOD>tatara main.as main.tro
Tatara MSX Macro-Assembler v1.2.0
Copyright (C) 2026 Javier Lavandeira
https://tatara.tools

ended at MAIN.AS(23)

A:\TATARA\EXAMPLES\TWOMOD>tatara putstr.as putstr.tro
Tatara MSX Macro-Assembler v1.2.0
Copyright (C) 2026 Javier Lavandeira
```

```
https://tatara.tools

ended at PUTSTR.AS(32)
```

The two object files could be named on TANREN's command line, as `HELLO.TRO` was. This example uses a *link file* instead: a text file that holds what would otherwise be typed on the command line. `TWOMOD.LNK` names the program and the two object files:

```
; TWOMOD.LNK - the link, in a file instead of on the command line.
;
; TANREN reads this when it is given @TWOMOD, and the words in it are
; the words it would have read from the command line. A ; starts a
; comment and the rest of the line is ignored.
;
; MSX-DOS gives a command line 127 characters and cuts a longer one
; without saying so. That is the whole reason this exists: the
; assembler is itself linked from a file of nineteen module names.

/o:twomod.com

main.tro putstr.tro
```

A program with many modules needs a link file, because an MSX-DOS2 command line holds at most 127 characters (chapter 22). TANREN reads a link file when its name is given after an `@`:

MSX-DOS

```
A:\TATARA\EXAMPLES\TWOMOD>tanren @twomod
Tatara MSX Linker v1.2.0
Copyright (C) 2026 Javier Lavandeira
https://tatara.tools

2 modules, 14 records, ends at EOF.
Wrote TWOMOD.COM, 0100-0154 (85 bytes), entry 0100.
```

These are the numbers of table 2.3: `MAIN` at `0100h`, `PUTSTR` straight after it, and 85 bytes in all, ending at `0154h`. Run the program:

MSX-DOS

```
A:\TATARA\EXAMPLES\TWOMOD>twomod
Two modules, one program.
The linker joined them.
```

4.8 What is next

Part II describes the assembler in full, starting with how to run it (chapter 5). Part IV has a worked guide to each of the other examples.

PART II

The assembler, Tatara

5 Running TATARA 33

6 Source format 43

7 Names 49

8 Expressions 55

9 Instructions 61

10 Segments and the location counter 69

11 Data 77

12 Defining and sharing symbols 81

13 Including files 87

14 The include files 95

15 Conditional assembly 103

16 Macros 111

17 Repeat blocks 121

18 The listing 127

19 The symbol table dump and the diagnostic switches 137

20 Error messages 147

CHAPTER 5

Running TATARA

5.1 The command line 33
5.2 Filenames 34
5.3 The options 36
5.4 What TATARA prints 37
5.5 Listings 38
5.6 When TATARA does not start 39

Chapter 4 ran TATARA in one way: **`tatara hello.as hello.tro`**. This chapter describes the whole command line: the filenames it takes, what happens when one is left out, every option, and everything TATARA prints on its own behalf. The sessions use `HELLO` and run in `A:\TATARA\EXAMPLES\HELLO`.

5.1 The command line

A TATARA command has this form:

```
tatara [options] source [object [listing]]
```

The words in square brackets can be left out. `source` is the file to assemble, `object` the object file to write, and `listing` a file for the listing, which section 5.5 describes.

An option is a slash followed by one letter, such as `/Q`. The letter can be typed in upper or lower case. Options can go anywhere on the line, before, between or after the filenames, and each one is a word of its own: write **`/l /s`**, not **`/ls`**. TATARA reads only the first letter after a slash and ignores the rest of the word, so **`/ls`** is the same as **`/l`**.

5.2 Filenames

The filenames are taken in order: the first is the source file, the second the object file, and the third the listing file. A fourth is an error (section 5.6).

5.2.1 No extension is added

TATARA uses each name exactly as it is typed, and adds no extension of its own. The extensions in table 5.1 are the usual ones, and the examples use them, but they have to be typed.

File	Usual extension
source file	`.AS` (`.ASM` and `.Z80` are also common)
object file	`.TRO`
listing file	`.LST` or `.PRN`

Table 5.1: The usual extensions.

Warning. **`tatara hello`** looks for a file named `HELLO`, with no extension, and stops if there is none:

MSX-DOS

```
A:\TATARA\EXAMPLES\HELLO>tatara hello
Tatara MSX Macro-Assembler v1.2.0
Copyright (C) 2026 Javier Lavandeira
https://tatara.tools

ERROR: cannot open HELLO
```

A name can include a drive and a directory, as any MSX-DOS2 filename can: **`tatara src\main.as obj\main.tro`** reads the source from the directory `SRC` and writes the object file to `OBJ`.

5.2.2 Leaving out the object file

Without a second filename, TATARA reads and assembles the source in the same way, reports any errors, and writes nothing. It is a quick way to find out whether a source has errors:

MSX-DOS

```
A:\TATARA\EXAMPLES\HELLO>tatara hello.as
Tatara MSX Macro-Assembler v1.2.0
Copyright (C) 2026 Javier Lavandeira
https://tatara.tools

ended at HELLO.AS(22)
```

5.2.3 When the object file is written

TATARA reads the source twice, and creates the object file only after the first reading has gone through the whole source. An error found during the first reading stops TATARA before that, so it leaves an object file of the same name as it was. Here `BAD.AS` holds a single line that is not an instruction, and `HELLO.TRO` is the same file, with the same time, before and after:

MSX-DOS

```
A:\TATARA\EXAMPLES\HELLO>dir hello.tro
 Volume in drive A: is MAIN
 Directory of A:\TATARA\EXAMPLES\HELLO

HELLO    TRO        93 26-09-29  2:15
 93 bytes in 1 file   12384K free
A:\TATARA\EXAMPLES\HELLO>tatara bad.as hello.tro
Tatara MSX Macro-Assembler v1.2.0
Copyright (C) 2026 Javier Lavandeira
https://tatara.tools

BAD.AS(1): ERROR: not a directive, a macro or an instruction.

A:\TATARA\EXAMPLES\HELLO>dir hello.tro
 Volume in drive A: is MAIN
 Directory of A:\TATARA\EXAMPLES\HELLO

HELLO    TRO        93 26-09-29  2:15
 93 bytes in 1 file   12376K free
```

The error message names the file and the line, in brackets, and says what is wrong. Chapter 20 describes the messages.

5.2.4 The object file cannot be the source

If the second filename is the same as the first, TATARA refuses to start, because writing the object file would destroy the source:

MSX-DOS

```
A:\TATARA\EXAMPLES\HELLO>tatara hello.as hello.as
Tatara MSX Macro-Assembler v1.2.0
Copyright (C) 2026 Javier Lavandeira
https://tatara.tools

ERROR: the output file is the input file.
```

TATARA compares the two names as they were typed. It does not notice the same file written in two ways, such as **`hello.as`** and **`a:hello.as`**, so take care with the order of the names.

5.3 The options

Table 5.2 lists the options for everyday use. Each has a chapter of its own, or is described in this one.

Option	What it does	Chapter
`/P`	prints a listing on the screen	18
`/L`	writes a listing to the file named third	18
`/S`	prints the symbol table when the assembly ends	19
`/C`	makes symbol and macro names case-sensitive	7
`/Q`	leaves out the banner and the summary line	5
`/V`	prints the banner, and stops	5
`/?`	prints the usage screen, and stops	5

Table 5.2: TATARA's options.

`/C` changes one thing: without it, `Start`, `START` and `start` are the same name, as they are in M80; with it, they are three different names. Instructions, registers and directives can be written in either case with or without `/C`.

5.3.1 Options for diagnosis

Three more options appear on the usage screen. They show what TATARA made of a source, and are meant for tracking down a problem in TATARA itself rather than for everyday use.

/F Instead of listing each line, prints the four fields TATARA divided it into (chapter 6), each in square brackets.

/M Prints every macro definition as TATARA stored it.

/H Prints, at the end, how many blocks of the memory TATARA keeps for macros are still in use.

5.4 What TATARA prints

5.4.1 The banner and the summary line

Every run that assembles a source starts with the *banner*: the program's name and version, the copyright, and the web address, followed by a blank line. It ends with the *summary line*, which says where the source ended: the file and the line of its `END`, or its last line if it has none. Chapter 4 showed both. If `END` is in an include file, the summary line names that file (chapter 13).

`/Q` leaves out both. Errors are printed with or without `/Q`, so a run with `/Q` that prints nothing is one that succeeded:

MSX-DOS

```
A:\TATARA\EXAMPLES\HELLO>tatara /q hello.as hello.tro

A:\TATARA\EXAMPLES\HELLO>
```

`/V` prints the banner and stops without assembling anything, even if a source is named:

MSX-DOS

```
A:\TATARA\EXAMPLES\HELLO>tatara /v
Tatara MSX Macro-Assembler v1.2.0
Copyright (C) 2026 Javier Lavandeira
https://tatara.tools


A:\TATARA\EXAMPLES\HELLO>
```

5.4.2 The usage screen

`/?` prints the banner and the *usage screen*, a summary of the command line, and stops. Typing **`tatara`** with no filename at all prints the same screen.

MSX-DOS

```
A:\TATARA\EXAMPLES\HELLO>tatara /?
Tatara MSX Macro-Assembler v1.2.0
Copyright (C) 2026 Javier Lavandeira
https://tatara.tools

Usage:  TATARA [options] <source> [output]
        TATARA /P [options] <source> [output]
        TATARA /L [options] <source> <output> <listing>

Options:  /P Assemble to the screen
          /L Write a listing file, named third on the line
          /S Print the symbol table when the assembly ends
          /C Make symbol and macro names case-sensitive
          /F Print the fields of each line, for diagnosis
          /M Print every macro definition, for diagnosis
          /H Print how many heap blocks are still in use
          /Q Omit the banner above and the summary line
          /V Print the program version
          /? Print this screen

<source>  : The file to assemble, M80 syntax (.as, .asm or .z80)
<output>  : The relocatable object it produces, usually .tro
<listing> : The listing file, usually .lst or .prn
```

The screen fills the 24 lines of the MSX display, so the line with the command scrolls off the top.

5.5 Listings

A *listing* is a printout of the source as TATARA assembled it, divided into pages with a heading on each. TATARA produces one when `/P` or `/L` is given, or when a third filename is named.

`/P` prints the listing on the screen. The session below shows only its beginning:

MSX-DOS

```
A:\TATARA\EXAMPLES\HELLO>tatara /p hello.as
Tatara MSX Macro-Assembler v1.2.0
Copyright (C) 2026 Javier Lavandeira
https://tatara.tools

        Tatara v1.2.0   29-Sep-26       PAGE    1


; HELLO.AS - the smallest complete program.
;
; One source, one object, one .COM. It prints a line and gives the
; machine back to MSX-DOS.
```

Each page starts with a form feed, which clears the MSX screen, so a listing longer than one page goes past faster than it can be read. `/P` is useful for a quick look at a short source.

`/L` writes the listing to the file named third on the line, with the address and the bytes of each line in front of it:

MSX-DOS

```
A:\TATARA\EXAMPLES\HELLO>tatara /l hello.as hello.tro hello.lst
Tatara MSX Macro-Assembler v1.2.0
Copyright (C) 2026 Javier Lavandeira
https://tatara.tools

ended at HELLO.AS(22)
```

The listing includes the lines of every file the source includes, and ends with a table of all the symbols, so even `HELLO.LST` is about 20 KB long: most of it is `MSXDOS.INC`. Chapter 18 describes what a listing contains and how a source controls it.

5.6 When TATARA does not start

Table 5.3 lists the messages TATARA prints when it stops before assembling a single line. Some of them come before the banner, because TATARA checks the command line first.

Two of them look like this:

MSX-DOS

Message	Meaning
`ERROR: Tatara needs MSX-DOS2 or Nextor.`	The MSX is running MSX-DOS1.
`ERROR: Tatara needs a memory mapper.`	The MSX has no memory mapper (chapter 1).
`ERROR: unknown option.`	A slash is followed by a letter that is not one of TATARA's options, or by nothing.
`ERROR: too many filenames.`	More than three filenames were given.
`ERROR: the output file is the input file.`	The object file has the same name as the source.
`ERROR: cannot open` *name*	The source file does not exist, or its name was typed without the extension.
`ERROR: cannot create` *name*	The object file cannot be written, for instance because the disk is full or write-protected.

Table 5.3: Messages before assembly.

```
A:\TATARA\EXAMPLES\HELLO>tatara /x hello.as
ERROR: unknown option.

A:\TATARA\EXAMPLES\HELLO>tatara a b c d
ERROR: too many filenames.
```

CHAPTER 6

Source format

6.1 Lines 43
6.2 The four fields 44
6.3 Where a label goes 44
6.4 Comments 46
6.5 Upper and lower case 46
6.6 Spaces, tabs, line length and line endings 46

This chapter describes how a source file is laid out: what a line is, the four parts TATARA divides it into, and the rules for labels, comments, upper and lower case, spaces and tabs, line length and line endings. It is the chapter to come back to when TATARA reports that a line is "not a directive, a macro or an instruction" and the line looks right.

6.1 Lines

A source file is a text file, and TATARA reads it one line at a time. Each line is one of three things:

- an empty line, which TATARA skips;
- a comment line, which holds only a comment for the reader;
- a *statement*: an instruction for the Z80, a directive for TATARA, or the name of a macro (chapter 16).

You can write a source file with any editor that saves plain text, either on the MSX or on another computer. Section 6.6 explains why a file written on a PC or a Mac can be assembled as it is.

6.2 The four fields

TATARA divides every line into up to four *fields*, always in the same order:

```
label      operation  operand       ; comment
```

Figure 6.1 shows a line with all four.

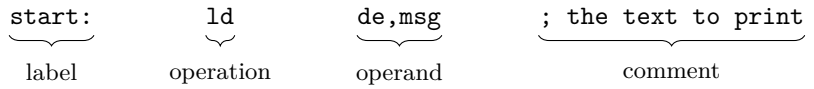


Figure 6.1: A source line and its four fields.

Label. A name for the address of the line (chapter 2), such as `start:`. The next section says where a label can be written, and chapter 7 what a name can contain.

Operation. What the line does: an instruction such as `ld`, a directive such as `db`, or the name of a macro. It ends at the first space or tab after it. This field is called the *operation*.

Operand. What the operation works on, such as `de,msg`. It runs from the operation to the comment, or to the end of the line, and can contain spaces: in `db "Hello from Tatara."` the whole of `"Hello from Tatara."` is the *operand*. What an operand can hold depends on the operation (chapters 8 to 11).

Comment. Text for the reader, from a semicolon to the end of the line. TATARA ignores it.

Every field can be left out. Table 6.1 shows the shapes a line usually takes.

6.3 Where a label goes

TATARA recognises a label in one of two ways:

1. A word that starts in column 1, the first character of the line, is a label, with or without a colon after it.
2. A word that does not start in column 1 is a label only if a colon ends it.

Line	Fields
`start:`	label
`start:  ld de,msg`	label, operation, operand
`    ret`	operation
`    ld c,_TERM0`	operation, operand
`    jp BDOS ; back to MSX-DOS2`	operation, operand, comment
`; HELLO.AS - the smallest complete program.`	comment
(empty)	none

Table 6.1: The usual shapes of a line.

An indented word without a colon is therefore the operation. In this source, `INDENT2.AS`, `start` is indented and has no colon, so TATARA takes it for an operation, and there is none of that name:

```
  start   ld      a,1
          end
```

MSX-DOS

```
INDENT2.AS(1): ERROR: not a directive, a macro or an instruction.
```

With a colon after `start`, the same indented line is accepted.

Warning. Anything that starts in column 1 is a label, including an instruction. In this source, `COL1.AS`, the instruction was typed without an indent:

```
ld      a,1
        end
```

TATARA takes `ld` for a label and `a,1` for the operation, and stops:

MSX-DOS

```
COL1.AS(1): ERROR: not a directive, a macro or an instruction.
```

When this message names a line that looks correct, check that the line starts with a space or a tab.

A label written with two colons, as in `putstr::`, is also made public, which is what the `PUBLIC` directive does (chapter 12).

The examples follow one convention, and it is a good one to copy: a label starts in column 1 and ends with a colon, and everything else on the line, and every line without a label, is indented with tabs.

6.4 Comments

A *comment* starts with a semicolon and runs to the end of the line. A line can be a comment and nothing else, and then the semicolon can be in column 1, as in the first lines of `HELLO.AS`.

A semicolon inside a string does not start a comment. In this line the string is `"a;b"`, and the comment starts at the second semicolon:

```
        db      "a;b"           ; the semicolon is part of the string
```

A quotation mark that is not closed on the same line does not start a string. This matters for the one Z80 instruction that has an apostrophe in it:

```
        ex      af,af'          ; swap the two AF registers
```

6.5 Upper and lower case

Instructions, register names and directives can be written in upper or lower case, or in a mixture: `LD A,B`, `ld a,b` and `Ld A,b` are the same line.

Names are not case-sensitive either, unless TATARA is run with `/C` (chapter 5): without it, `msg`, `MSG` and `Msg` are the same name. Chapter 7 says more.

The text inside a string keeps its case: `"Hello"` and `"HELLO"` are different strings.

The examples write their code in lower case, and the include files write their names in upper case, as the MSX documentation does: `ld c,_STROUT`.

6.6 Spaces, tabs, line length and line endings

6.6.1 Spaces and tabs

Spaces and tabs separate the fields, and any number of either, in any mixture, can be used between two fields. Tabs keep the fields in straight columns, and the examples use them for that reason. In a listing, each tab moves to the next column that is a multiple of eight (chapter 18).

6.6.2 Line length

A line can be up to 255 characters long. A longer line stops TATARA with this message:

MSX-DOS

```
ERROR: source line too long.
```

6.6.3 Line endings

Every text file marks the end of each line with one or two control characters. MSX-DOS and Windows use two, CR and LF; macOS and Linux use LF alone. TATARA ends a line at LF and ignores CR, so it reads both kinds of file, and a source written on another computer can be copied to the MSX and assembled without being converted. The last line of a file is read even if it has no line ending.

A file that uses CR alone, as some very old Macintosh programs write, is read as one long line, and TATARA stops with the message above.

6.6.4 Ctrl-Z

Some MSX-DOS editors put a Ctrl-Z, the character `1Ah`, after the last line of a file, a habit inherited from CP/M. TATARA stops reading at a Ctrl-Z, and ignores anything that comes after it.

CHAPTER 7

Names

7.1 The characters of a name 49
7.2 Length 50
7.3 Upper and lower case 51
7.4 Names that are hard to reach 53

A program is full of *names*: labels, values defined with `EQU`, macros, segments. Chapter 2 called them symbols, and chapter 12 describes how each kind gets its value. This chapter is about how a name is spelled: the characters it can contain, how long it can be, whether upper and lower case matter, and which names are best avoided.

7.1 The characters of a name

A name is made of the characters in table 7.1, in any order, except that it must not start with a digit.

Characters	
`A` to `Z`, `a` to `z`	letters
`0` to `9`	digits, but not first
`? @ . _ $`	five other characters

Table 7.1: The characters a name can contain.

These are the characters M80 allows. All four names in this source are ordinary labels:

```
        cseg
?tmp:           nop
@loop:          nop
my.name:        nop
a_b$c:          nop
```

Four names that M80 allows cannot be names in TATARA: `IXH`, `IXL`, `IYH` and `IYL`, which are registers (section 9.5).

Any other character in a label stops TATARA with `a name may hold only letters, digits, ? @ . _ and the dollar sign.`

The underscore is the one most programs use, to join the words of a long name: `print_string`. The others are legal, but a reader may take `my.name` or `a_b$c` for something other than a name, so the examples do not use them.

A word that starts with a digit is a number (chapter 8). TATARA does accept `1st:` as a label, but nothing can refer to it: `jp 1st` reads `1st` as a badly written number, and stops:

MSX-DOS

```
DIGIT2.AS(3): ERROR: bad expression.
```

7.2 Length

The name of a symbol, a label or a value defined with `EQU` or `DEFL`, can be up to 255 characters long, and every character counts: two names that differ only in their last character are two different symbols. This source, from the example `NAMES` (chapter 31), defines two symbols whose names differ only in the last five characters:

```
first_pass_symbol_table_high_water_mark_in_bytes        defl    1024
first_pass_symbol_table_high_water_mark_in_words        defl    512
```

M80 kept only the first six characters of a name, which is why so many programs from the CP/M era are written in abbreviations such as `PRTSTR`. With TATARA there is no need for them.

Symbol names keep their full length in the object file, so a public name in one module and the same name declared external in another are matched in full when TANREN links them (chapter 2).

> Note. A source line is at most 255 characters long too (chapter 6), so a 255-character name fills a line on its own, with no room for a colon. A name used after an instruc-

tion has less room: after a tab, `jp` and another tab, 251 characters are left. Names of a few dozen characters are long enough for any program.

The names of macros, segments and groups are shorter. A macro name can be up to 64 characters long, and a segment or group name (chapter 10) up to 16. A longer name is refused, not cut short, with one of these messages:

- `a macro name may be at most 64 characters.`
- `a segment or group name may be at most 16 characters.`

7.3 Upper and lower case

By default, as in M80, upper and lower case do not matter in a name: `Counter`, `counter` and `COUNTER` are the same name. The option `/C` (chapter 5) makes them three different names.

`/C` applies only to the names you choose. Instructions, registers and directives can be written in either case with or without it (chapter 6).

The example `NAMES` shows the difference. It defines two symbols that differ only in case, with `DEFL`, which allows a name to be given a new value:

```
; NAMES.AS - names as long as you like, in either case.
;
; A SYMBOL MAY BE 255 CHARACTERS AND ALL OF THEM COUNT. M80 keeps the
; first six, which is why so much CP/M-era assembly is written in
; abbreviations. The two names below differ only in their last five
; characters and are two different symbols here. 255 is the limit, and
; the test suite has a 240-character name in it; these are 48, which is
; as long as a line can carry and still be read on an MSX screen.

first_pass_symbol_table_high_water_mark_in_bytes        defl    1024
first_pass_symbol_table_high_water_mark_in_words        defl    512

; THE SAME SIX LETTERS IN TWO CASES. Without /C they are ONE symbol,
; and the second line changes its value. With /C they are TWO, with
; values 1 and 2. The symbol table of each run says which - and DEFL
; is used rather than EQU because a redefinition is legal, so the
; source assembles cleanly in both modes.

Counter         defl    1
counter         defl    2

; DIRECTIVES, MNEMONICS AND REGISTER NAMES ARE CASE-INSENSITIVE IN
; BOTH MODES. /C is about the names you choose, not the ones the Z80
```

```
; already has.

                cseg

start:          LD      A,1
                ld      b,a
                DB      Counter
                db      counter
                ret

                end     start
```

Assembled without `/C`, `Counter` and `counter` are one symbol, and the second line gives it a new value, 2. This is the end of the symbol table that `/S` prints (chapter 19):

MSX-DOS

```
A:\TATARA\EXAMPLES\NAMES>tatara /q /p /s names.as
...
ASEG - absolute

0002h  var        Counter
0200h  var        first_pass_symbol_table_high_water_mark_in_words
0400h  var        first_pass_symbol_table_high_water_mark_in_bytes
```

The table shows the name as it was written where it was first defined. With `/C` there are two symbols, with the values 1 and 2:

MSX-DOS

```
A:\TATARA\EXAMPLES\NAMES>tatara /q /p /c /s names.as
...
ASEG - absolute

0001h  var        Counter
0002h  var        counter
0200h  var        first_pass_symbol_table_high_water_mark_in_words
0400h  var        first_pass_symbol_table_high_water_mark_in_bytes
```

Use `/C` only when a program needs names that differ only in case, for instance because it was written for, or produced by, a tool that tells them apart. Otherwise the default is safer: a name typed in the wrong case still refers to the right symbol.

Warning. All the modules of a program must be assembled in the same way, all with `/C` or all without it. The object file records which it was, and TANREN refuses to link a mixture, because a name could then be found in one module and missed in another. Here `MAIN.AS` of `TWOMOD` was assembled with `/C` and `PUTSTR.AS` without it:

MSX-DOS

```
A:\TATARA\EXAMPLES\TWOMOD>tanren @twomod
Tatara MSX Linker v1.2.0
Copyright (C) 2026 Javier Lavandeira
https://tatara.tools

ERROR: one module was assembled /C and another was not.
```

7.4 Names that are hard to reach

TATARA has no reserved words: it defines any name it is given. But in some places it looks a word up in a fixed list before it looks for a name, and there a name spelled like a word in the list is never reached. There are four such places.

The operation field. TATARA looks for a directive first, then a macro, then an instruction. A macro named like a directive can never be called, and a macro named like an instruction is used instead of it (chapter 16).

An instruction's operand. If the first word of an operand is exactly the name of a register, the word is that register. The word ends at a space, a tab, a comma, a bracket, `+` or `-`, so `c` and `c+0` both start with the register C, while `bad` is a word of its own and an ordinary name. The registers are `A`, `B`, `C`, `D`, `E`, `H`, `L`, `BC`, `DE`, `HL`, `SP`, `AF`, `AF'`, `IX`, `IY`, `I` and `R`.

A condition. After `JP`, `JR`, `CALL` and `RET`, the conditions `NZ`, `Z`, `NC`, `C`, `PO`, `PE`, `P` and `M` come first.

An expression. `NOT`, `HIGH` and `LOW` are always operators, and `MOD`, `SHR`, `SHL`, `EQ`, `NE`, `LT`, `LE`, `GT`, `GE`, `AND`, `OR` and `XOR` are operators wherever an operator can stand (chapter 8). A symbol named `high` is defined, but `db high` stops with `bad expression.`

The name `c` shows most of this at once, because it is a register, a condition and a legal name. This source defines it, and uses it four times:

```
c               equ     5
                cseg
                ld      a,c
                ld      a,0+c
                jp      c,c
                db      c
```

Line	Bytes	What c was
`ld a,c`	`79`	the register
`ld a,0+c`	`3E 05`	the symbol, 5
`jp c,c`	`DA 05 00`	the condition, then the symbol
`db c`	`05`	the symbol

Table 7.2: Four uses of a symbol named `c`.

Table 7.2 shows the bytes TATARA writes for each line, from its listing.

Written `ld a,c+0`, the second line fails, because the operand starts with the register name:

MSX-DOS

```
REGNAME.AS(4): ERROR: not a form this instruction has.
```

The safe rule is simple: do not give a name the spelling of a register, a condition, an operator, a directive or an instruction. Appendix A lists the directives and appendix B the instructions.

7.4.1 Names that start with two question marks

The directive `LOCAL` (chapter 16) makes up names for the labels inside a macro, so that each time the macro is used its labels are different. The names are `??0000`, `??0001` and so on, and they appear in the symbol table like any other name:

MSX-DOS

```
CSEG - default code segment, 0009h bytes

0000h             start
0002h             ??0000
0006h             ??0001
```

Do not define names of that form yourself, or they may clash with the ones `LOCAL` makes.

7.4.2 The dollar sign on its own

`$` on its own is not a name but the location counter, the address of the current line (chapters 8 and 10). Inside a name, as in `a_b$c`, it is an ordinary character.

CHAPTER 8

Expressions

8.1 Values are 16 bits 55
8.2 Numbers 56
8.3 Characters 56
8.4 Names and the location counter 57
8.5 Operators 57
8.6 What kind of value 59

Wherever an instruction or a directive expects a number, you can write an *expression*: a number, a name, or a calculation made of them, such as `10`, `msg`, `msg+2` or `(count-1)*4`. TATARA works out its value when it assembles the line. This chapter describes how numbers and characters are written, what the operators do and in what order, and what kind of value an expression can have. The examples use `DW`, which stores the value of each expression as two bytes (chapter 11).

8.1 Values are 16 bits

Every value TATARA calculates is a 16-bit number, from 0 to 65535 (`0FFFFh`). Arithmetic that goes past either end wraps round: `65535+1` is 0, and `0-1` is `0FFFFh`. A negative number is therefore stored as its 16-bit complement, as the Z80 itself treats it: `-1` is `0FFFFh`, and `-6` is `0FFFAh`.

Where a line needs only one byte, such as the operand of `ld a,...` or of `DB`, TATARA uses the low byte of the value. Chapters 9 and 11 say which values are accepted there.

8.2 Numbers

A number is decimal unless a letter after it names another base, as in table 8.1. The letter can be upper or lower case.

Base	Suffix	Example	Value
decimal	none, or `D`	`100`, `100d`	100
hexadecimal	`H`	`100h`	256
octal	`O` or `Q`	`100o`, `100q`	64
binary	`B`	`100b`	4

Table 8.1: Numbers in each base.

A number always starts with a digit; a word that starts with a letter is a name (chapter 7). A hexadecimal number whose first digit is a letter therefore needs a `0` in front: `0ffh` is 255, but `ffh` is a name, and unless you have defined it TATARA stops with:

MSX-DOS

```
HEXNAME.AS(2): ERROR: undefined symbol in an expression.
```

> Warning. `B` and `D` are hexadecimal digits and also suffixes. A number that ends in one of them is read as binary or decimal: `10b` is 2, not 10Bh. Always end a hexadecimal number with its `H`, as in `0abh` or `12bh`.

A digit that does not belong to the base, such as the `2` in `12b`, stops TATARA:

MSX-DOS

```
BADDIGIT.AS(2): ERROR: bad expression.
```

8.3 Characters

A *character constant* is one or two characters between apostrophes or quotation marks. Its value is the character's code: `'A'` and `"A"` are both `41h`. With two characters the first goes in the high byte, so `"AB"` is `4142h`.

The two kinds of quotation mark work the same way, which is useful when the character is one of them: `"'"` is an apostrophe. Inside a constant, the mark that

encloses it can also be written twice to stand for itself, so `''''` is an apostrophe too.

Strings of more than two characters are for `DB` (chapter 11).

8.4 Names and the location counter

A name in an expression stands for its value: the address of a label, or the number given to it with `EQU` or `DEFL` (chapter 12). A name can be used before the line that defines it, because TATARA reads the source twice: in this source, `jp later` is assembled correctly.

```
                cseg
                jp      later           ; defined below
later:          ret
```

A name that is defined nowhere stops TATARA:

MSX-DOS

```
UNDEF.AS(2): ERROR: undefined symbol in an expression.
```

`$` on its own stands for the *location counter*: the address of the line it is written in, in the current segment (chapter 10). A common use is `jr $`, which jumps to itself and waits forever.

8.5 Operators

Table 8.2 lists the operators. The word operators can be written in upper or lower case, and need a space or a bracket on each side: `a and b`, not `aandb`.

A few of them need a word more:

- The comparisons give `0FFFFh` for true and 0 for false, so their results can be combined with `AND`, `OR` and `NOT`. They compare unsigned numbers: `-1 gt 1` is true, because `-1` is `0FFFFh`.
- Division by zero is not an error. As in M80, the result is `0FFFFh`.
- `HIGH` and `LOW` apply to the value right after them: `high 1234h+1` is `(high 1234h)+1`, which is `13h`.

Operator	What it does	Example and value
`+ -`	addition, subtraction	`10-4-3` = 3
`* /`	multiplication, division	`17/5` = 3
`MOD`	remainder of the division	`17 mod 5` = 2
`SHL SHR`	shift left, shift right	`1 shl 4` = `10h`
`AND OR XOR`	bitwise and, or, exclusive or	`0f0h and 3ch` = `30h`
`NOT`	inverts every bit	`not 0` = `0FFFFh`
`HIGH LOW`	high byte, low byte	`high 1234h` = `12h`
`EQ NE`	equal, not equal	`1 eq 1` = `0FFFFh`
`LT LE`	less than, less or equal	`2 le 1` = 0
`GT GE`	greater than, greater or equal	`2 gt 1` = `0FFFFh`
`-` (before a value)	negation	`-1` = `0FFFFh`

Table 8.2: The operators.

8.5.1 Precedence

When an expression has more than one operator, the *precedence* of each decides which is worked out first. Table 8.3 lists the levels, the first binding the tightest. Operators on the same level are worked out from left to right, so `10-4-3` is `(10-4)-3`. Brackets change the order: `2+3*4` is 14, and `(2+3)*4` is 20.

Level	Operators
1	`HIGH LOW`
2	`* / MOD SHL SHR`
3	`-` before a value
4	`+ -`
5	`EQ NE LT LE GT GE`
6	`NOT`
7	`AND`
8	`OR XOR`

Table 8.3: Operator precedence, tightest first.

These are M80's levels. Two consequences are worth knowing: `-2*3` is `-(2*3)`, and `not a eq b` is `not (a eq b)`. When in doubt, add brackets: they cost nothing.

8.6 What kind of value

Chapter 2 explained that TATARA cannot know every address, and leaves holes for TANREN to fill. Every expression therefore has a kind as well as a value:

Absolute. A number that is complete when TATARA assembles it, such as `10`, `'A'` or a name defined with `EQU`. Such a value is called *absolute*.

Relocatable. An address in a segment, such as a label in the code segment. TATARA knows only its distance from the start of the segment; TANREN adds the address where the segment is placed.

External. A name declared with `EXTRN`, which another module defines. TATARA knows nothing about its value; TANREN fills it in.

An expression is only useful if TANREN can finish it by adding one address, so TATARA allows only the combinations in table 8.4. Anything else needs both sides absolute.

Expression	Result	Example
absolute + relocatable, or the reverse	relocatable	`a1+2`
relocatable − absolute	relocatable	`a1-2`
relocatable − relocatable, same segment	absolute	`a2-a1`
external + absolute, or the reverse	external	`far+2`
external − absolute	external	`far-2`

Table 8.4: The combinations TATARA allows.

The difference of two labels in the same segment is the most useful of these: it is a length, and it does not depend on where the segment is placed. In this source `a2-a1` is 1, and `a1+2` is the address two bytes after `a1`, but `a1*2` has no meaning once the code is moved:

```
                cseg
a1:             nop
a2:             nop
                dw      a2-a1           ; absolute: 1
                dw      a1+2            ; relocatable
                dw      a1*2            ; not allowed
```

```
RELOC.AS(6): ERROR: relocation error - a segment-relative value is not allowed
    here.
```

An external name can be used only once in an expression, with a number added to it or taken away from it. `2-far` is refused:

MSX-DOS

```
EXTRN.AS(5): ERROR: an external symbol may not be used here.
```

Negation, `NOT`, `HIGH` and `LOW` need an absolute value too. Each segment is a kind of its own, so labels in two different segments cannot be subtracted from each other either (chapter 10).

CHAPTER 9

Instructions

9.1 The instruction set 61
9.2 Operands 62
9.3 Relative jumps 63
9.4 The R800 instructions 64
9.5 The undocumented instructions. 65
9.6 When an operand is wrong 68

This chapter describes how the Z80's instructions, and the two the R800 adds, are written for TATARA: the forms their operands can take, index registers and displacements, relative jumps, and the messages TATARA gives when an operand is wrong. It does not explain what each instruction does. That is the job of the book or course you learn Z80 programming from; appendix B lists the instructions by class, for reference.

9.1 The instruction set

TATARA accepts the Z80 instructions, written the way Zilog writes them, the *undocumented instructions* that Zilog never published but that every Z80 runs (section 9.5), and the R800's `MULUB` and `MULUW` (section 9.4): 70 mnemonics in all. The source that chapter 6 described puts the mnemonic in the operation field and its operands, separated by commas, in the operand field.

9.2 Operands

An operand is one of the forms in table 9.1. Which forms an instruction takes is set by the Z80, and your Z80 book lists them. The bytes in the table are the ones TATARA writes for each example.

Form	Example	Bytes
register	`ld a,b`	`78`
immediate value	`ld a,5`	`3E 05`
	`ld hl,1234h`	`21 34 12`
register indirect	`ld a,(hl)`	`7E`
	`ex (sp),hl`	`E3`
	`jp (hl)`	`E9`
indexed	`ld a,(ix+5)`	`DD 7E 05`
	`ld a,(ix-1)`	`DD 7E FF`
	`ld a,(iy)`	`FD 7E 00`
memory address	`ld a,(1234h)`	`3A 34 12`
	`ld hl,(1234h)`	`2A 34 12`
port	`in a,(98h)`	`DB 98`
	`out (c),a`	`ED 79`
condition	`jr nz,$`	`20 FE`
bit number	`set 0,(ix+3)`	`DD CB 03 C6`
fixed value	`rst 38h`	`FF`
	`im 1`	`ED 56`

Table 9.1: Operand forms.

9.2.1 Values and brackets

Wherever an instruction takes a number, an address or a port, the operand can be any expression (chapter 8). An operand without brackets is an *immediate value*: the number itself goes into the instruction. An operand in brackets is a memory address, or a port for `IN` and `OUT`. `ld a,5` loads the number 5 into A; `ld a,(5)` loads the byte stored at address 5.

An immediate value that occupies one byte, as in `ld a,5`, is the low byte of the expression. TATARA does not check that the value fits: `ld a,-1` writes `0FFh`,

which is what is meant, but `ld a,300` writes `2Ch`, the low byte of 300, without a message. One byte also cannot hold an address that TANREN still has to finish, so a one-byte value must be absolute (section 8.6). A two-byte value, as in `ld hl,msg`, can be relocatable or external.

9.2.2 Index registers

An indexed operand is `(IX+d)` or `(IY+d)`, where `d`, the *displacement*, is a signed number from −128 to 127 that is added to the index register. It can be written with `+` or `-`, and it can be any absolute expression: `(ix-1)`, `(ix+count*2)`. `(IX)` alone means `(IX+0)`. A displacement out of range stops TATARA:

MSX-DOS

```
DISP.AS(2): ERROR: an index displacement must be -128 to 127.
```

9.3 Relative jumps

`JR` and `DJNZ` are *relative jumps*: the instruction holds the distance to the target, one signed byte, rather than its address. In the source you still write the target, as with `JP`, and TATARA works out the distance: `jr nz,loop`, `djnz loop`.

The distance is measured from the instruction that follows, and must be from −128 to 127 bytes. `jr $`, which jumps to itself, is a distance of −2 and assembles to `18 FE`. A target further away stops TATARA:

MSX-DOS

```
JR.AS(2): ERROR: a JR or DJNZ can only reach -128 to 127.
```

The target must be in the same segment as the jump, or be an absolute address, because a distance to code that TANREN places elsewhere is not known until the program is linked. Use `JP` for a target that is too far away or in another module.

Instruction	Bytes	What it does
`mulub a,`*r*	`ED C1`+8*r*	A × *r*, the result in HL
`muluw hl,bc`	`ED C3`	HL × BC, the result in DE:HL
`muluw hl,sp`	`ED F3`	HL × SP, the result in DE:HL

Table 9.2: The R800 instructions.

9.4 The R800 instructions

The R800, the processor of the MSX turbo R, adds two multiplication instructions to the Z80's set. Table 9.2 shows the three forms TATARA accepts.

r is an 8-bit register: `mulub a,b` is `ED C1`. TATARA accepts any of them, but the R800's documentation guarantees the result only for `B`, `C`, `D` and `E`. The first operand must be `A` for `MULUB` and `HL` for `MULUW`.

> Warning. TATARA assembles these instructions on any MSX, but a program that uses them runs correctly only on a turbo R. On a Z80 they do not multiply.

9.4.1 R800 mnemonics

Every instruction is stored as a few bytes. The first byte, or the first two when the instruction has a prefix such as `DD` or `ED`, says which instruction it is; this part is the *opcode*. The rest, if there is any, holds the operand: a number, an address or a displacement. The processor reads only the bytes. A *mnemonic*, such as `LD` or `DJNZ`, is a name that people give to an opcode so that they can write programs without learning the numbers.

Because a mnemonic is only a name, the same opcodes can be given more than one set of names. The R800's documentation from ASCII, reprinted in the *MSX-Datapack*, gives the whole instruction set new mnemonics: `JP` is `br` (branch), `DJNZ` is `dbnz`, `EX` is `xch`, `RES` is `clr`, `LDIR` is `movem`. Register names start with a dot, square brackets stand for memory, and the accumulator is always written out: Zilog's `SUB B` is `sub .a,.b`. Apart from the two multiplications, every R800 instruction has the same opcode as the Z80 instruction it renames, and does the same thing.

Table 9.3 shows a short routine, which clears 256 bytes of memory, written both ways. The bytes are the same.

`ld b,0` makes `DJNZ` go round 256 times, because `B` reaches zero only after the 256th decrement.

	Z80	R800	Bytes
clear:	ld hl,buffer	ld .hl,buffer	21 *nn nn*
	ld b,0	ld .b,0	06 00
	xor a	xor .a,.a	AF
loop:	ld (hl),a	ld [.hl],.a	77
	inc hl	inc .hl	23
	djnz loop	dbnz loop	10 FC
	ret	ret	C9

Table 9.3: The same routine in Z80 and R800 mnemonics.

At the time of writing, TATARA accepts only the Z80 mnemonics. A program written in R800 mnemonics has to be translated to Z80 mnemonics before TATARA can assemble it; the program that results is the same.

9.5 The undocumented instructions

Zilog published the Z80's instructions, but not all of the opcodes the processor runs. Some of the others do useful things, are the same on every Z80, and have been used in MSX programs for decades. TATARA accepts four groups of them:

- `SLL`, a shift to the left that sets bit 0;
- the *index halves* `IXH`, `IXL`, `IYH` and `IYL`, the high and low bytes of `IX` and `IY` used as 8-bit registers;
- `IN F,(C)`, which reads a port and sets the flags without keeping the value;
- `OUT (C),0`, which writes a fixed value to a port.

Like the R800's instructions, they are assembled on any MSX, with no option to turn them on. M80 has none of them (appendix J). Table 9.4 shows their forms; appendix B lists all 104 with their bytes.

Form	Bytes
`sll` *r*, for B, C, D, E, H, L and A	`CB 30+`*r*
`sll (hl)`	`CB 36`
`sll (ix+`*d*`)`, `sll (iy+`*d*`)`	`DD CB` *d* `36`, `FD CB` *d* `36`
`ld` *x*`,`*r* and `ld` *r*`,`*x*, for B, C, D, E and A	`DD` for `IX`, `FD` for `IY`, then the byte of the same instruction with `H` for a high half and `L` for a low one: `ld ixh,b` is `DD` and the `60` of `ld h,b`
`ld` *x*`,`*y*, both halves of the same register	the same: `ld ixh,ixl` is `DD 65`
`ld` *x*`,`*n*	the same: `ld ixl,5` is `DD 2E 05`
`inc` *x*, `dec` *x*	the same: `inc iyh` is `FD 24`
`add a,`*x*, `adc a,`*x*, `sub` *x*, `sbc a,`*x*, `and` *x*, `xor` *x*, `or` *x*, `cp` *x*	the same: `cp ixl` is `DD BD`
`in f,(c)`	`ED 70`
`out (c),0`	`ED 71`

Table 9.4: The undocumented instructions. *r* is an 8-bit register, with the codes of appendix B; *x* and *y* are index halves.

9.5.1 How the index halves work

There is no opcode for `IXH`. A `DD` in front of an instruction makes it use `IX` where it would use `HL`, and that applies to the halves too: `H` becomes `IXH` and `L` becomes `IXL`. `ld a,h` is `7C`, so `ld a,ixh` is `DD 7C`. With `FD`, the same instructions use `IY`.

The prefix belongs to the whole instruction, not to one operand, and that decides which forms exist (figure 9.1):

- `ld h,ixl` cannot be written: `DD 65` turns both operands into halves, and is `ld ixh,ixl`.
- `ld ixh,iyl` cannot be written: an instruction can have only one of the two prefixes.
- `ld ixh,(hl)` cannot be written: with the prefix, `(HL)` becomes `(IX+`*d*`)`, and `DD 66` *d* is `ld h,(ix+`*d*`)`.

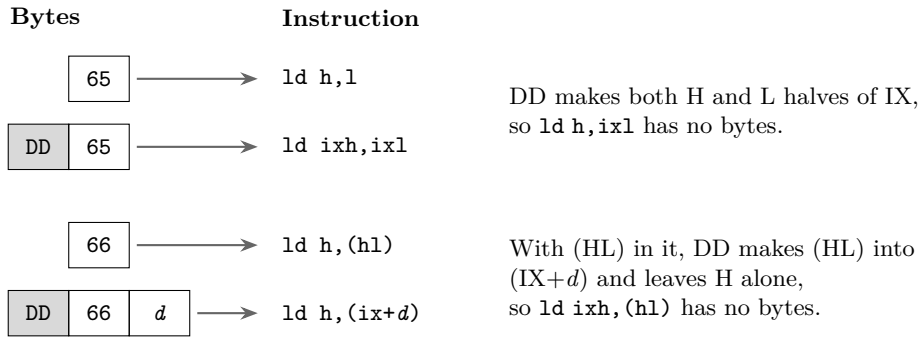


Figure 9.1: One prefix for the whole instruction: what DD does to two instructions.

The halves are also refused wherever the Z80 has no form for them: in the rotations and shifts, in `BIT`, `SET` and `RES`, in `IN` and `OUT`, and next to an indexed address, as in `ld (ix+5),ixh`. In each case TATARA stops with `not a form this instruction has.`

9.5.2 What a program may not rely on

Warning. **SLL on the MSX turbo R.** The R800 runs `CB 30` to `CB 37`, but as `SLA`: bit 0 is left clear. With `B` = 81h, `sll b` leaves 03h on a Z80 and 02h on an R800. The carry is the same on both, so only bit 0 shows the difference. A program that uses `SLL` does not run the same on a turbo R. The index halves, `IN F,(C)` and `OUT (C),0` work on the R800 as they do on the Z80.

Warning. **OUT (C),0 does not always write 0.** The NMOS Z80 writes 0; the CMOS Z80 writes 0FFh, and so do both processors in openMSX. A program cannot rely on the value it writes.

`IN F,(C)` leaves `A` alone and sets the flags as `IN A,(C)` would. Its only use is to test a port's bits with the flags.

9.5.3 Spellings and names

Each of these instructions has one spelling: `SLL`, not `SL1` or `SLIA`; `IXH` and the others, not `HX`, `LX`, `HY` or `LY`; `IN F,(C)`, not `IN (C)`; and `OUT (C),0` with the digit 0 itself, not `00h` or an expression. The other spellings give the usual messages for a word TATARA does not know.

IXH, IXL, IYH and IYL are registers, so they can no longer be names. M80 allows them as labels, and in TATARA such a label would silently change what `ld a,ixh` means, so TATARA refuses it, and refuses them in PUBLIC and EXTRN too:

MSX-DOS

```
IXHNAM.AS(2): ERROR: IXH, IXL, IYH and IYL are registers, not names.
```

HX, LX, HY and LY stay ordinary names, so a program that uses them for coordinates assembles as before.

Two other groups of undocumented opcodes are not accepted: the forms such as `rlc (ix+2),b`, which store the result in a register as well as in memory, and the extra opcodes in the ED group that repeat NEG, RETN and NOP.

9.6 When an operand is wrong

Table 9.5 lists the messages TATARA gives for an operand it cannot use, each with a line that causes it.

Line	Message
`ld hl,a`	`not a form this instruction has.`
`mulub b,c`	`not a form this instruction has.`
`ld a,(ix+200)`	`an index displacement must be -128 to 127.`
`jr far`	`a JR or DJNZ can only reach -128 to 127.`
`rst 5`	`RST takes 0, 8, 10h and so on to 38h.`
`im 3`	`IM takes 0, 1 or 2.`
`bit 8,a`	`a bit number must be 0 to 7.`

Table 9.5: Messages about operands.

`not a form this instruction has.` is the general one: the instruction exists, but not with these operands. `ld hl,a` is an example, because the Z80 cannot copy an 8-bit register into a 16-bit one. Check the forms the instruction has in your Z80 book, or in appendix B.

CHAPTER 10

Segments and the location counter

10.1 The location counter 69
10.2 Code, data and absolute 70
10.3 ORG 71
10.4 Reserving space: DS 71
10.5 Named segments 72
10.6 Transient data segments 73
10.7 Messages 75

Chapter 2 introduced segments: areas of a program that hold one kind of content, code or data, and that TANREN places in memory separately. This chapter describes them in full: how TATARA keeps count of addresses in each segment, the directives that choose a segment, set its address and reserve space in it, the named segments that let a program keep related code or data together, and transient data segments, a feature of Tatara's own that lets variables share memory.

10.1 The location counter

As TATARA assembles a segment, it counts the bytes each line produces (chapter 2). The count is the *location counter*. A label takes the value the location counter has at its line, and `$` reads it in an expression (chapter 8).

Each segment has a location counter of its own. When a source changes segment, TATARA puts the current counter aside and takes up the other segment's; when it comes back, it continues from where it stopped. In this source the code and the data are written in turns, and each kind is counted separately:

```
                cseg
start:          ld      hl,count
                inc     (hl)
                dseg
count:          ds      1
                cseg
again:          ret
                dseg
total:          ds      2
```

TATARA's listing shows the address of each line:

MSX-DOS

```
 0000'   21 00 00            start:          ld      hl,count
 0003'   34                                  inc     (hl)
                                             dseg
 0000'                       count:          ds      1
                                             cseg
 0004'   C9                  again:          ret
                                             dseg
 0001'                       total:          ds      2
```

`again` is at offset 4 in the code segment, straight after `inc (hl)`, and `total` at offset 1 in the data segment, straight after `count`. The apostrophe after an address means that it is relocatable: an offset in its segment, which TANREN finishes (chapter 2). The listing uses the same mark for code and data; chapter 18 describes it.

10.2 Code, data and absolute

Table 10.1 lists the three kinds of segment and the directive that selects each. A source starts in the code segment, so a program with no data of its own needs no directive at all.

Directive	Segment	Its addresses
`CSEG`	code segment	relocatable: TANREN places it
`DSEG`	data segment	relocatable: TANREN places it
`ASEG`	absolute segment	fixed, as written

Table 10.1: The three kinds of segment.

Keeping code and data apart lets TANREN place the data somewhere other than straight after the code: a ROM cartridge, for instance, must keep its variables

in RAM (chapter 33). By default TANREN places all the code segments first and the data segments after them; chapter 24 describes the order in full and the options that choose other addresses.

In the *absolute segment* every address is exactly the one written, and TANREN does not move it. It is for code or data that must be at a known address, such as the header of a ROM cartridge at `4000h`.

> Warning. A `.COM` program is loaded at `0100h` and is one continuous block of bytes. An absolute address far from there, such as `4000h`, makes the file cover everything in between, so it becomes about 16 KB long. Use the absolute segment for `.COM` programs only when you know why you need it.

10.3 ORG

`ORG` sets the location counter of the current segment. In the absolute segment its operand is an address; in the code and data segments it is an offset from the start of the segment.

```
                aseg
                org     4000h
rom:            db      'AB'            ; at address 4000h
                cseg
                org     10h
code10:         nop                     ; 10h bytes into the code
```

The operand of `ORG` must be known when TATARA first reaches the line, so it cannot use a name defined further down. It must also be absolute, or an address in the current segment: an address in the data segment cannot be the location counter of the code segment, and TATARA stops with the relocation error of chapter 8.

10.4 Reserving space: DS

`DS` (or `DEFS`, its other name) moves the location counter on by a number of bytes, and so reserves space for variables:

```
                dseg
buffer:         ds      128             ; 128 bytes
count:          ds      1               ; 1 byte
```

The count must be absolute and known on the first reading, like the operand of `ORG`. `DS` takes only the count: a second operand, such as a value to fill the space

with, is a `bad expression.` `DS` writes no bytes: it only moves the location counter, so a program should give its variables a value before it reads them.

10.5 Named segments

A program can have more than one code segment and more than one data segment. A *named segment* is selected by writing a name after `CSEG` or `DSEG`:

```
                cseg    music           ; all the music code
                dseg    strings         ; all the text
                dseg    variables       ; all the variables
```

A named segment behaves like the plain one of its kind, with a location counter of its own, and every line that selects the same name adds to the same segment. Code for one purpose can therefore be written in several places, or in several source files, and still end up in one piece. TANREN joins the segments of the same name from every module it links.

In this source the code segment and the segment `music` are each selected twice:

```
                cseg
main:           nop
                cseg    music
play:           nop
                nop
                cseg
more:           nop
                cseg    music
stop:           ret
```

The symbol table (chapter 19) shows each segment with its size, and each label in the segment it belongs to:

MSX-DOS

```
CSEG - default code segment, 0002h bytes

0000h             main
0001h             more

music - named code segment, 0003h bytes

0000h             play
0002h             stop
```

Each segment is a kind of value of its own (section 8.6), because TANREN may place two segments anywhere relative to each other. The difference between labels

in two different segments is therefore refused, with the relocation error.

10.6 Transient data segments

Most programs have variables that are used by only one part of the program. A routine that reads a line typed by the user needs a buffer for it while it reads; a routine that prints a report needs a buffer for the line it is building. If the two routines never run at the same time, the two buffers are never in use at the same time either, and they could occupy the same memory. On an MSX, where a `.COM` program has less than 64 KB for its code and its data together, the saving can matter.

Other assemblers leave this to the programmer, who has to work out the shared addresses by hand, usually with `EQU`, and keep them right every time a variable changes size. Tatara does it for you, with *transient data segments*.

10.6.1 How it works

A transient data segment is a named data segment with the word `TRANSIENT` after its name. Its variables are divided into *groups*, each started by a `GROUP` line with a name:

```
                dseg    scratch,transient
                group   reading         ; used only while reading
inbuf:          ds      64
inlen:          ds      2
                group   writing         ; used only while writing
outbuf:         ds      32
outlen:         ds      2
```

The rules are few:

- Within a group, variables follow one another as in any data segment, and never overlap. `inbuf` and `inlen` are both usable at the same time.
- Every group starts at the beginning of the segment. `outbuf` is at the same address as `inbuf`.
- The segment is as large as its largest group.
- Every variable must be in a group: a `DS` line before the first `GROUP` is an error.

Figure 10.1 shows the four variables of the example in a plain data segment and in the transient one. In the plain segment they take 100 bytes; in the transient

one they take 66, the size of the group `reading`, and the 34 bytes of `writing` cost nothing.

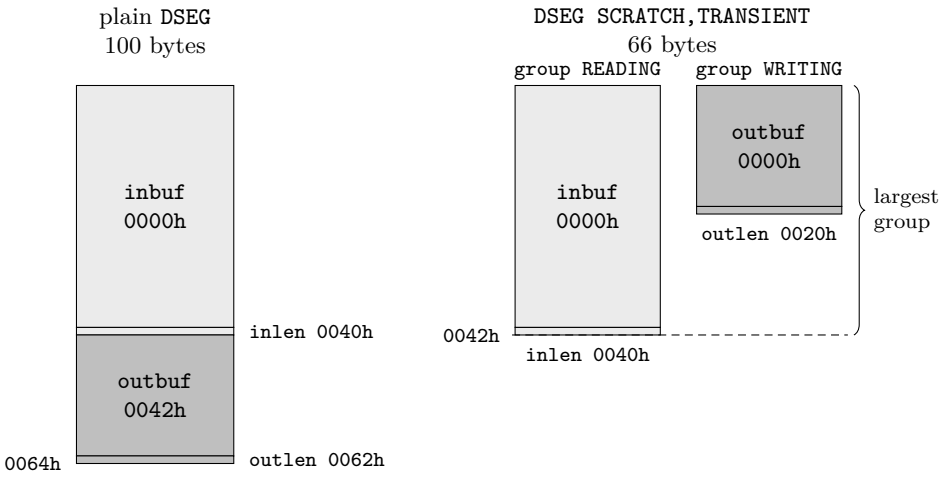


Figure 10.1: The same four variables in a plain data segment and in a transient one, drawn to scale.

TATARA's symbol table shows the segment and each group, with its size and the offset of each variable in it:

MSX-DOS

```
scratch - named data segment, transient, group reading, 0042h bytes

0000h             inbuf
0040h             inlen

scratch - named data segment, transient, group writing, 0022h bytes

0000h             outbuf
0020h             outlen
```

10.6.2 Using it safely

A group says which variables are in use together. Everything else is the programmer's responsibility: TATARA does not follow the program to check that two groups are never in use at the same time.

Warning. A variable in one group has the same address as the variables at the same offset in every other group of the segment. Writing to one destroys the others. If the routine that uses `writing` called the routine that uses `reading` while `outbuf` still held a line it had not printed, the line typed by the user would be written over it. Put variables that must survive each other's use in the same group, in different transient segments, or in a plain data segment.

Two points complete the rules:

- Only the groups of one transient segment overlap each other. Different segments never share memory, so a program can have one transient segment for each set of routines that take turns.
- Group names are global: a group of the same name in the same segment of another module is the same group, and TANREN joins them. Chapter 29 describes a program in several modules that uses this.

Labels in two different groups cannot be subtracted from each other, because neither of them says anything about the other's position. TATARA gives the relocation error.

10.7 Messages

Table 10.2 lists the messages about segments, each with a line that causes it.

A program can have a limited number of segments, groups and external names; going past it gives `too many segments, groups or externals.` Appendix E lists the limits.

Line	Message
`dseg vars,forever`	`bad ASEG, CSEG or DSEG line.`
`dseg` and a name of 17 characters	`a segment or group name may be at most 16 characters.`
`dseg vars,transient`, after `dseg vars`	`this segment was declared differently before.`
`group oops` in the code segment	`GROUP needs a name, no label, and a transient DSEG.`
`x: ds 1` before the first `GROUP`	`this transient DSEG needs a GROUP first.`
`dw second-first`, labels in two groups or two segments	`relocation error - a segment-relative value is not allowed here.`
`org v`, `v` in another segment	`relocation error - a segment-relative value is not allowed here.`

Table 10.2: Messages about segments.

CHAPTER 11

Data

11.1 DB: bytes 77
11.2 Strings 78
11.3 DW: words 78
11.4 DC: strings that mark their own end 79
11.5 The other names 79
11.6 Messages 80

A program needs data as well as instructions: messages to print, tables of numbers, addresses. This chapter describes the directives that put data into a program: `DB` for bytes and strings, `DW` for 16-bit words, and `DC` for strings that mark their own end. `DS`, which reserves space without filling it, is in chapter 10.

11.1 DB: bytes

`DB` takes a list of expressions separated by commas, and stores each one as a byte:

```
table:          db      1,2,3           ; 01 02 03
                db      'A'+1           ; 42
                db      -1              ; FF
```

Each expression can be anything chapter 8 describes. As with instructions (chapter 9), TATARA stores the low byte of the value and does not check that it fits: `db 300` stores `2Ch`.

One byte cannot hold an address that TANREN still has to finish, so the value must be absolute. A label or an external name in `DB` stops TATARA with the relocation error:

MSX-DOS

```
DBREL.AS(3): ERROR: relocation error - a segment-relative value is not allowed
    here.
```

11.2 Strings

A *string* in `DB` is text between quotation marks, and is stored as one byte for each character. Strings and numbers can be mixed in one list:

```
msg:            db      'Hello',13,10,'$'
```

These are the bytes `48 65 6C 6C 6F 0D 0A 24`: the five letters, then 13 and 10, which move to the start of the next line, then the `$` that the MSX-DOS2 function `_STROUT` stops at (chapter 4).

A string can be enclosed in apostrophes or in quotation marks. Inside it, the enclosing mark is written twice to stand for itself, or the other mark can be used instead. These two lines store the same four bytes:

```
                db      "It's"
                db      'It''s'
```

An empty string, `''`, stores nothing.

> Note. A string of one or two characters looks the same as a character constant (chapter 8), and TATARA tells them apart as M80 does, by what follows the closing quotation mark. If a comma or the end of the line follows, the item is a string; otherwise it is a character constant in an expression. `db 'AB'` is therefore two bytes, `41 42`, but `db 'AB' and 0ffh` is one: the value `4142h` with its high byte removed, `42h`.

11.3 DW: words

`DW` stores each expression as a *word*: two bytes, low byte first, which is the order in which the Z80 stores every 16-bit value (chapter 2). `dw 1234h` is `34 12`.

Two bytes can hold an address that TANREN finishes, so a label, an expression built on one, or an external name is allowed here (section 8.6). This makes `DW` the way to build a table of addresses, such as a list of routines to jump to:

```
jumps:          dw      start, again, finish
```

A string of one or two characters in `DW` is a character constant: `dw 'AB'` is `42 41`, the value `4142h` stored low byte first. A longer string stops TATARA with `bad DB, DW or DC operand.`

11.4 DC: strings that mark their own end

`DC` takes one string, and stores it like `DB` except that bit 7 of the last character is set. `dc 'Stop'` is `53 74 6F F0`: the last byte is the `p`, `70h`, plus `80h`.

The characters of ordinary text all have bit 7 clear, so a program can recognise the last character of such a string by that bit, and the string needs no terminator such as `$`. This routine prints a string made with `DC`, whose address is in `HL`. It uses the MSX-DOS2 function `_CONOUT`, which prints the character in `E`, through the `system` macro of `MSXDOS.INC` (chapter 14):

```
print:          ld      a,(hl)          ; the next character
                inc     hl
                push    af              ; keep it, and its bit 7
                and     7fh             ; the character without bit 7
                ld      e,a
                push    hl              ; MSX-DOS2 may change HL
                system  _CONOUT
                pop     hl
                pop     af
                bit     7,a             ; was it the last one?
                jr      z,print         ; no: go round again
                ret
```

If `AND`, `BIT` and the stack are new to you, the book you learn Z80 programming from explains them; what matters here is that the loop ends by itself, on the byte whose bit 7 is set.

`DC` must be given exactly one string, and not an empty one, because an empty string has no last character to mark. `dc ''` and `dc 5` both stop TATARA with `bad DB, DW or DC operand.`

11.5 The other names

Some directives have a second name, which TATARA accepts so that sources written for other assemblers can be assembled unchanged. Table 11.1 lists them.

Other name	Same as
`DEFB, DEFM`	`DB`
`DEFW`	`DW`
`DEFS`	`DS`

Table 11.1: Other names of the data directives.

11.6 Messages

Table 11.2 lists the messages about data, each with a line that causes it.

Line	Message
`db 1,,2`	`bad DB, DW or DC operand.`
`dw 'ABC'`	`bad DB, DW or DC operand.`
`dc '',dc 5`	`bad DB, DW or DC operand.`
`db label,db far` (external)	`relocation error - a segment-relative value is not allowed here.`

Table 11.2: Messages about data.

CHAPTER 12

Defining and sharing symbols

12.1 Three ways to give a name a value 81
12.2 Labels 82
12.3 EQU 82
12.4 DEFL 83
12.5 Sharing names between modules 83
12.6 END and the start address 85
12.7 Messages 86

A symbol is a name that stands for a number (chapter 2). This chapter describes the three ways a name gets its value, the rule that a name has only one value, the directives that let modules share names, and `END`, which ends a source and names where the program starts.

12.1 Three ways to give a name a value

Table 12.1 lists them.

Written as	Its value	Can change
`name:`	the location counter at that line	no
`name equ` *value*	the value given	no
`name defl` *value*	the value given	yes

Table 12.1: Three ways to give a name a value.

A name for a number makes a program easier to read and to change. `ld c,_STROUT` says what it does, where `ld c,9` does not; and if the number ever

changes, only the line that defines the name has to change with it. The names in `MSXDOS.INC` are defined with `EQU` for exactly this reason (chapter 14).

12.2 Labels

A label takes the value of the location counter at the line where it is written: the address of the first byte that line produces (chapter 10). A label is defined once. A second label of the same name stops TATARA:

MSX-DOS

```
LABTWICE.AS(3): ERROR: this name already has a value.
```

12.3 EQU

`EQU` gives a name a fixed value, the value of the expression after it:

```
five            equ     5
screen          equ     4000h
here            equ     $               ; an address in the code
```

The name is written without a colon: with one, TATARA takes it for a label, and the `EQU` is left with no name. `five: equ 5` stops TATARA with `EQU, DEFL and MACRO take a name, not a label.`

The value can be any expression (chapter 8), including an address, as `here` shows: it is relocatable, like a label in the same place. It cannot be an external name, which has no value until the program is linked.

A name defined with `EQU` can be used before the line that defines it. In this source, `later` is used on the line before its `EQU`, and TATARA assembles it correctly as `0Ah`:

```
                ld      de,later
later           equ     five*2
```

`EQU` gives a name its value for good. Defining the same name again with the same value is accepted, but with a different value TATARA stops with `this name already has a value.`

12.4 DEFL

`DEFL` also gives a name a value, but the name can be given a new value later, as often as the source needs. Each line that uses the name sees the value it has at that point in the source:

```
count           defl    0
                db      count           ; 00
count           defl    count+1
                db      count           ; 01
count           defl    count+1
                db      count           ; 02
```

A name defined with `DEFL` is sometimes called a *variable symbol*, and the symbol table marks it `var`. It cannot share its name with a label or an `EQU`: `n equ 1` followed by `n defl 2` stops TATARA with `this name already has a value.` Like `EQU`, it takes a name without a colon.

`DEFL` is most useful in conditional assembly and in macros, as a counter or a switch that changes as the source is read (chapters 15 to 17).

> Note. Readers who know M80 from other computers may expect a `SET` directive as another name for `DEFL`. Neither M80 in Z80 mode nor TATARA has one, because `set` is the Z80 instruction that sets a bit: `x set 5` is read as that instruction, and stops TATARA with `not a form this instruction has.` `DEFL` is the only symbol that can be redefined. (A macro can be redefined too; that is a different matter, described in chapter 16.)
> In the same way, TATARA has no `.Z80` directive. M80 needed it to accept Z80 mnemonics; TATARA accepts nothing else, and has no mode to switch. A source written for M80 that starts with `.Z80` stops on that line with `not a directive, a macro or an instruction.`, and the line can simply be deleted.

12.5 Sharing names between modules

Chapter 2 explained that a program can be made of several modules, and that a module offers names to the others with `PUBLIC` and uses theirs with `EXTRN`. Each takes a list of names separated by commas, and each has a second name: `ENTRY` for `PUBLIC`, and `EXT` for `EXTRN`. A label written with two colons, as in `both::`, is made public as well (chapter 6).

This module offers four names, one written each way:

```
                public  greet,value
                entry   other
                cseg
greet:          ret
other:          ret
both::          ret
value           equ     5
```

`PUBLIC` can come before the definition, as it does here, or after it. Labels and `EQU` values can be public, and so can a `DEFL` name. The symbol table marks each public name:

MSX-DOS

```
ASEG - absolute

0005h  public     value

CSEG - default code segment, 0003h bytes

0000h  public     greet
0001h  public     other
0002h  public     both
```

This module uses them:

```
                extrn   greet,value
                ext     other,both
                cseg
start:          call    greet
                call    other
                call    both
                ld      hl,value
                ret
                end     start
```

TATARA knows nothing about an external name except that another module defines it, so the symbol table lists the four under `EXTERNAL - resolved by the linker`. An external value is finished by TANREN, so it can go only where two bytes hold it, as in `ld hl,value`; in one byte, such as `ld a,value`, it is refused (section 8.6). Linked together, the two modules give a complete program:

MSX-DOS

```
A:\>tanren /o:use.com use.tro pub.tro
Tatara MSX Linker v1.2.0
Copyright (C) 2026 Javier Lavandeira
https://tatara.tools

2 modules, 20 records, ends at EOF.
```

```
Wrote USE.COM, 0100-010F (16 bytes), entry 0100.
```

Two mistakes are caught by TATARA itself:

- A name declared `EXTRN` and then defined in the same module stops TATARA with `this name already has a value.`: it cannot be both.
- A name declared `PUBLIC` and never defined stops TATARA with `a PUBLIC name was never defined.` This check is made when TATARA writes the object file; without an object file name on the command line, the source is accepted.

Mistakes that involve more than one module, such as a name that no module defines, are found by TANREN (chapter 27).

12.6 END and the start address

`END` ends the source. TATARA reads nothing after it, not even the rest of a file that included the one containing it (chapter 13). A source with no `END` is accepted too: it ends where the file ends.

The operand of `END`, if there is one, names the *start address* of the program: where it begins to run. TATARA writes it into the object file, and TANREN shows it as `entry` in its summary line (chapter 4).

Warning. MSX-DOS2 always starts a `.COM` program at `0100h`, the first byte of the file, whatever `END` says. In a `.COM` program, the label named by `END` must therefore be at the very start of the code. If data comes first, as in this source, the program runs the data as if it were instructions and crashes. TANREN writes the file all the same, but warns that the start address is not where the program will be entered:

```
                cseg
msg:            db      'Started at start.',13,10,'$'
start:          ld      de,msg
                ...
                end     start
```

MSX-DOS

```
WARNING: ENTRY.COM is entered at 0100, not at 0114.
1 modules, 7 records, ends at EOF.
Wrote ENTRY.COM, 0100-0120 (33 bytes), entry 0114.
```

When TANREN gives this warning for a `.COM` program, move the code named by `END` to the start, or put the data in the data segment (chapter 10).

12.7 Messages

Table 12.2 lists the messages about defining and sharing symbols, each with a line that causes it.

Line	Message
`five: equ 5`	`EQU, DEFL and MACRO take a name, not a label.`
`five equ 6` after `five equ 5`; a label twice; `n defl 2` after `n equ 1`; a name both `EXTRN` and defined	`this name already has a value.`
`x equ far`, `far` external	`an external symbol may not be used here.`
`public` with no names	`bad PUBLIC or EXTRN list.`
`public nowhere`, never defined	`a PUBLIC name was never defined.`
`x set 5`	`not a form this instruction has.`

Table 12.2: Messages about symbols.

One more message belongs here, although a correct source never causes it:

MSX-DOS

```
phase error - this label had a different value on pass 1.
```

TATARA reads the source twice, and a label must get the same value both times. Chapter 20 describes when it can happen.

CHAPTER 13

Including files

13.1 INCLUDE 87
13.2 Where TATARA looks 89
13.3 The TATARA environment variable 90
13.4 Paths in the filename 90
13.5 Files that include files 90
13.6 Including a file twice 91
13.7 END in an include file 92
13.8 Messages 92

A source can take lines from another file with `INCLUDE`. This chapter describes what `INCLUDE` does, where TATARA looks for the file, how the `TATARA` environment variable fits in, and what happens when files include other files.

13.1 INCLUDE

`include` followed by a filename reads that file at that point, as if its lines had been typed there instead of the `INCLUDE` line. When the file ends, TATARA carries on with the line after the `INCLUDE`. A file read this way is called an *include file*.

```
; DEFS.INC - two names for MAIN.AS
five            equ     5
ten             equ     10
```

```
                include defs.inc        ; the names
                cseg
                ld      a,five
                ld      b,ten
                ret
                end
```

The listing shows the lines of `DEFS.INC` where the `INCLUDE` line was, with nothing to mark them as coming from another file. The `INCLUDE` line itself is not listed:

MSX-DOS

```
A:\>tatara /l /p main.as
Tatara MSX Macro-Assembler v1.2.0
Copyright (C) 2026 Javier Lavandeira
https://tatara.tools

        Tatara v1.2.0   29-Sep-26       PAGE    1


                                ; DEFS.INC - two names for MAIN.AS
  0005                          five            equ     5
  000A                          ten             equ     10
                                                cseg
  0000'   3E 05                                 ld      a,five
  0002'   06 0A                                 ld      b,ten
  0004'   C9                                    ret
                                                end
ended at MAIN.AS(6)
```

The filename is written as it is, with its extension and without quotation marks, and a comment can follow it. TATARA adds nothing to the name: `include defs` looks for a file called `DEFS`, and `include "defs.inc"` for a file whose name starts with a quotation mark. Both stop TATARA with `cannot open`.

An include file can hold anything a source can: names, code, data, macros, and `INCLUDE` lines of its own. The usual use is a set of names that several programs share, such as the names of the MSX-DOS2 functions in `MSXDOS.INC` (chapter 14).

Including a file is not the same as linking a module (chapter 2). An include file becomes part of the source that includes it, and of the one object file TATARA makes from it. Its names are that source's names, and nothing needs `PUBLIC` or `EXTRN`.

13.2 Where TATARA looks

A filename with no drive and no directory, such as `DEFS.INC`, can be in more than one place. TATARA looks for it in three places, in this order, and reads the first one it finds:

1. the directory of the file that contains the `INCLUDE` line;
2. the *current directory*, where you were when you typed the command;
3. each directory named in the `TATARA` environment variable, from left to right.

Figure 13.1 shows the three places for one command. There is a `DEFS.INC` in each of them, and TATARA reads the one in `SRC`, next to `MAIN.AS`.

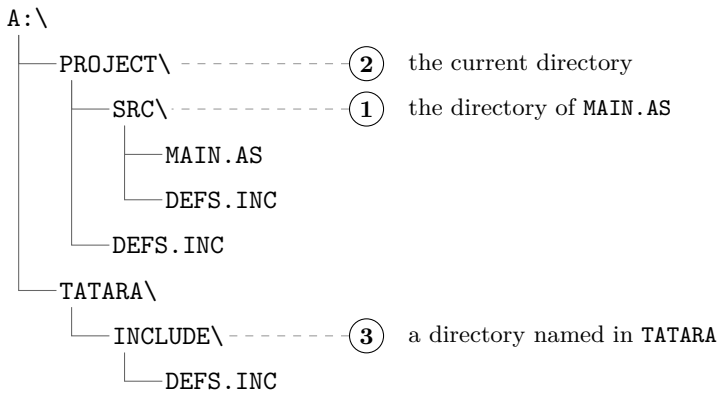


Figure 13.1: The three places TATARA looks for an include file, in order.

The directory of the including file comes first so that a project's own include file always wins over another file of the same name in the current directory or in the `TATARA` directories. In figure 13.1, `A:\PROJECT\DEFS.INC` is in the current directory, but TATARA never reaches it: the copy in `SRC` is found first.

The current directory is still searched, second. If `SRC` had no `DEFS.INC`, TATARA would read the one in `A:\PROJECT`, the directory the command was typed in.

> Note. The file named on the command line is not searched for. TATARA opens it as you typed it, from the current directory unless you name another (chapter 5). The three places apply only to `INCLUDE`.

13.3 The TATARA environment variable

Chapter 3 set TATARA to the directory of the include files that come with Tatara. It can name several directories, separated by semicolons, and TATARA tries them from left to right. In AUTOEXEC.BAT or another batch file:

```
set tatara=a:\tatara\include;a:\project\inc
```

At the prompt, the value goes between quotation marks, as in chapter 3.

A \ at the end of each directory is not needed, but does no harm.

The environment variable is the last of the three places. With the line above, a file that is in both directories comes from A:\TATARA\INCLUDE, the first one named. A file that is also in the current directory, or in the directory of the including file, comes from there, and the environment variable is not used at all.

13.4 Paths in the filename

The filename after INCLUDE can name a directory too.

A name that starts with a drive, such as A:\TATARA\INCLUDE\MSXDOS.INC, or with \, such as \TATARA\INCLUDE\MSXDOS.INC, is an *absolute path*: it says exactly where the file is. TATARA opens it as written and looks nowhere else.

Any other name is a *relative path*, and is looked for in each of the three places in turn. include sub\mid.inc looks for SUB\MID.INC in the directory of the including file, then in the current directory, then under each directory in TATARA.

On a Japanese MSX, \ appears as ¥, as in chapter 3. It is the same character.

13.5 Files that include files

An include file can include another. TATARA can have four files open at once: the source, and three levels of include files. A fourth level stops TATARA, and the message shows how it got there, one line for each level, innermost first:

```
D2.INC(2): ERROR: too many source files open at once.
    included from D1.INC(2)
    included from D0.INC(2)
    included from DEEPBAD.AS(2)
```

Here `DEEPBAD.AS` included `D0.INC`, which included `D1.INC`, which included `D2.INC`, and line 2 of `D2.INC` asked for a fifth file. A file that includes itself runs into the same limit.

Macros have a limit of their own, described in chapter 16.

13.6 Including a file twice

A file that holds only `EQU`s and macros can be included any number of times: the second time, each name gets the same value again, which is allowed (chapter 12), and a macro can be defined again (chapter 16).

A file that defines a label cannot. The second copy defines the label again:

MSX-DOS

```
CODE.INC(2): ERROR: this name already has a value.
    included from TWICELAB.AS(2)
```

Include a file that holds labels or code in one place only.

Warning. An *include guard* is a way some programmers make a file safe to include twice: the file's contents go inside `IFNDEF`, which skips them when a name is already defined, and the first line inside defines that name. It does not work in TATARA.

```
                ifndef  gcode_inc
gcode_inc       equ     1
                cseg
shared:         ret
                endif
```

TATARA reads the source twice (chapter 5), and a name defined during the first reading is still defined during the second. So on the second reading `gcode_inc` is already defined, and `IFNDEF` skips the file even where it is included only once. TATARA notices that the label `shared` was not defined the second time, and stops:

MSX-DOS

```
GCODE.INC(4): ERROR: this label was defined on pass 1 and not on pass 2 -
    a conditional skipped it.
```

TATARA's messages call each reading a *pass*. Chapter 15 describes conditional assembly and the two passes.

13.7 END in an include file

`END` in an include file ends the whole source, not only that file (chapter 12). The lines after the `INCLUDE` are never read, and the summary line says where the source ended:

MSX-DOS

```
ended at ENDIT.INC(2)
```

An include file should not contain `END`.

13.8 Messages

Table 13.1 lists the messages about including files, each with a line that causes it.

Line	Message
`include` with no name	`INCLUDE without a filename.`
`include nothere.inc`, a file that is in none of the three places	`cannot open NOTHERE.INC`
a fifth file open at once, or a file that includes itself	`too many source files open at once.`

Table 13.1: Messages about including files.

When TATARA cannot open an include file, the message has no file and line in front of it. The lines below it say where the file was asked for:

MSX-DOS

```
ERROR: cannot open NOTHERE.INC
    included from MIDMISS.INC(2)
    included from NESTMISS.AS(2)
```

Any other error inside an include file starts with that file's name and line, followed by the same trail:

MSX-DOS

```
BAD.INC(2): ERROR: not a form this instruction has.
    included from MIDBAD.INC(2)
    included from ERRINC.AS(3)
```

Chapter 20 describes how to read a trail, including the lines it adds for macros.

> Note. M80 also accepts `MACLIB` and `$INCLUDE` as other names for `INCLUDE`. TATARA does not. In a source written for M80, change them to `INCLUDE`.

CHAPTER 14

The include files

14.1 What the files hold 95
14.2 Including them 96
14.3 MSXDOS.INC and the system macro 97
14.4 BIOS.INC and SUBROM.INC 98
14.5 ERRORS.INC 99
14.6 The other files 100

Tatara comes with nine include files of names for the MSX: the MSX-DOS2 functions and error codes, the BIOS and SUB ROM entry points, the hooks, the system work area, the I/O ports, the extended BIOS and the control codes. Chapter 3 put them in `A:\TATARA\INCLUDE`, and chapter 13 showed how `INCLUDE` finds them. This chapter describes what is in them and what to know before including each one. Appendix K summarises each file and names the books that describe what is in it.

14.1 What the files hold

Table 14.1 lists the nine files. Together they hold 903 names and one macro, `system`, which is in `MSXDOS.INC`.

The names are the ones the MSX documentation uses, as the MSX Datapack spells them, so a name in your source is the same as the name in a book or in another program's listing. The exceptions are `PORTS.INC`, because the MSX has no standard names for its I/O ports, and the prefixes in `SUBROM.INC` and `EXTBIO.INC` (sections 14.4 and 14.6).

Apart from the macro, every name is an `EQU`. An `EQU` gives a name a value and puts nothing in the program, so including a file adds names, not bytes. A source

File	What it holds	Names
`ASCII.INC`	the control codes	37
`BIOS.INC`	the MAIN ROM entry points	92
`ERRORS.INC`	the MSX-DOS and MSX-DOS2 error codes	76
`EXTBIO.INC`	the extended BIOS, and the MSX-MUSIC FM BIOS	85
`HOOKS.INC`	the hooks	117
`MSXDOS.INC`	the MSX-DOS function numbers, and the `system` macro	93
`PORTS.INC`	the I/O ports	56
`SUBROM.INC`	the SUB ROM entry points	36
`WORKAREA.INC`	the system work area	311

Table 14.1: The include files.

that includes all nine files and holds a single `ret` is still one byte long:

MSX-DOS

```
A:\>tanren /o:nocost.com nocost.tro
Tatara MSX Linker v1.2.0
Copyright (C) 2026 Javier Lavandeira
https://tatara.tools

1 modules, 5 records, ends at EOF.
Wrote NOCOST.COM, 0100-0100 (1 bytes).
```

Each file starts with a comment that says how to use what is in it, and each name has a comment of one line that says what it is. What a BIOS routine or an MSX-DOS2 function needs in its registers, and what it gives back, is in the MSX books; the files do not repeat it, and neither does this manual.

14.2 Including them

Include a file by its name, as in chapter 4:

```
                include msxdos.inc
```

The names of an included file become names of your source. A label of your own cannot have the same name as one of them. `beep:` in a source that includes `BIOS.INC`, where `BEEP` is a BIOS entry, stops TATARA:

MSX-DOS

```
CLASH.AS(4): ERROR: this name already has a value.
```

Include only the files a source uses, and it will not meet names it does not need.

A file of names can be included more than once, in one source or in every module of a program, because each name gets the same value again (chapter 13).

The names are in upper case. TATARA does not tell upper and lower case apart unless the `/C` option is given (chapter 7), so `ld a,(linl40)` is the same as `ld a,(LINL40)`. With `/C`, write the names as they are in the files:

MSX-DOS

```
CASEC.AS(4): ERROR: undefined symbol in an expression.
```

14.3 MSXDOS.INC and the system macro

`MSXDOS.INC` holds the numbers of the MSX-DOS functions, with the underscore that the MSX-DOS2 documentation gives them, such as `_STROUT` and `_TERM0`, and the name `BDOS` for 0005h, the address a program calls to reach them.

A program puts the function number in `C`, anything else the function needs in the other registers, and calls `BDOS`. The `system` macro does the first and the last of these in one line:

```
start:          ld      de,msg
                system  _STROUT
                system  _TERM0
msg:            db      'Printed with system.',13,10,'$'
```

The listing shows what TATARA makes of each `system` line: the two instructions of the macro, each marked `+`.

MSX-DOS

```
0000'   11 0D 00              start:          ld      de,msg
                                              system  _STROUT
0003'   0E 09           +                     ld      c,_STROUT
0005'   CD 05 00        +                     call    BDOS
                                              system  _TERM0
0008'   0E 00           +                     ld      c,_TERM0
000A'   CD 05 00        +                     call    BDOS
```

`system` changes `C`, and whatever the function itself changes. It always calls 0005h, which is right for a `.COM` program under MSX-DOS. A program that runs under Disk BASIC, or from a cartridge, reaches the functions at 0F37Dh instead, and has to make the call itself.

The functions below 31h exist in MSX-DOS1 as well. Those from 31h up need MSX-DOS2.

14.4 BIOS.INC and SUBROM.INC

`BIOS.INC` holds the entry points of the MAIN ROM, in address order.

> Warning. Under MSX-DOS, the MAIN ROM is not in memory at 0000h to 3FFFh: that page is RAM. A plain `call` to a BIOS entry from a `.COM` program therefore does not reach the BIOS. Five entries are the exception, because MSX-DOS provides them at the same addresses: `RDSLT`, `WRSLT`, `CALSLT`, `ENASLT` and `CALLF`. Every other entry needs an *inter-slot call*, through `CALSLT`.

`CALSLT` takes the slot of the routine in the high byte of `IY`, and its address in `IX`. The slot of the MAIN ROM is the first byte of `EXPTBL`, in the work area, so `ld iy,(EXPTBL-1)` loads it into the high byte of `IY`. This program sounds the buzzer, then prints a line with `CHPUT`, one character at a time:

```
                include msxdos.inc
                include bios.inc
                include workarea.inc
                cseg
start:          ld      iy,(EXPTBL-1)   ; the MAIN ROM slot, into IYh
                ld      ix,BEEP
                call    CALSLT
                ld      hl,msg
next:           ld      a,(hl)
                or      a
                jr      z,done
                push    hl
                ld      iy,(EXPTBL-1)
                ld      ix,CHPUT
```

```
                call    CALSLT
                pop     hl
                inc     hl
                jr      next
done:           ei
                system  _TERM0
msg:            db      'Printed by the BIOS.',13,10,0
                end     start
```

MSX-DOS

```
A:\>bioscall
Printed by the BIOS.
```

The `ei` at the end turns interrupts back on, since an inter-slot call leaves them off.

`SUBROM.INC` holds the entry points of the SUB ROM, which the MSX2 added. The SUB ROM uses some of the MAIN ROM's names for different routines at different addresses: `GRPPRT` is 008Dh in the MAIN ROM and 0089h in the SUB ROM. So every name in `SUBROM.INC` starts with `SUBROM_`, as in `SUBROM_GRPPRT`, and the two files can be included together.

A SUB ROM routine is reached through the BIOS entry `EXTROM`, with its address in `IX`:

```
                ld      ix,SUBROM_REDCLK
                call    EXTROM
```

That is the way from Disk BASIC or from a cartridge. From MSX-DOS it is more awkward: `EXTROM` is itself a BIOS entry, reached through `CALSLT`, and `CALSLT` uses `IX` for the address it calls, so it cannot hand a SUB ROM address to `EXTROM` in `IX`. The comment after `EXTROM` in `BIOS.INC` describes the way round it.

14.5 ERRORS.INC

`ERRORS.INC` holds the error codes of MSX-DOS and MSX-DOS2. Their names start with a dot, as in the MSX-DOS2 documentation: `.NOFIL` is 0D7h, "File not found". A dot is allowed in a name (chapter 7). The codes count down from 0FFh, so this file, unlike the others, is in descending order.

An MSX-DOS2 function numbered 40h or higher returns an error code in `A`, and zero if it worked. A `jr nz` right after the call catches an error. A program can then hand the code to MSX-DOS2 with the function `_TERM`, which ends the

program and prints the message for that code. This program tries to open a file that is not there:

```
                include msxdos.inc
                cseg
start:          ld      de,name
                xor     a               ; open mode 0
                system  _OPEN
                jr      nz,failed       ; A is the error code
                system  _TERM0
failed:         ld      b,a
                system  _TERM
name:           db      'NOTHERE.TXT',0
                end     start
```

MSX-DOS

```
A:\>openerr

*** File not found
```

To test for one error in particular, compare `A` with its name: `cp .NOFIL`.

14.6 The other files

14.6.1 HOOKS.INC

A *hook* is five bytes in RAM that the system calls at a fixed moment: `H.TIMI`, for instance, is called at every timer interrupt, 50 or 60 times a second. An unused hook holds a `ret`. A program that takes over a hook saves its five bytes first, writes a jump to its own routine in their place, and puts the five bytes back before it ends. A hook left pointing into a program that has ended crashes the MSX at the next call.

Two hooks have two names each, because MSX-MIDI renamed them: `H.ONKO` and `H.MDIN` are both 0FF75h, and `H.FRQI` and `H.MDTM` are both 0FF93h. They are the same five bytes, not four hooks.

14.6.2 WORKAREA.INC

The *work area* is the part of RAM, from 0F323h up, where the system keeps its variables. In `WORKAREA.INC` the number at the start of each comment is the size of the variable in bytes. Read a one-byte variable with `ld a,(LINL40)`, and a two-byte one with `ld hl,(...)`; reading one with the other's size gives an answer that is nearly right, and wrong.

14.6.3 PORTS.INC

The names of the I/O ports are Tatara's own, since the MSX has no standard ones. Every port name starts with `IO_`, as in `IO_VDPCMD` for port 099h, and is written with three hex digits, the leading zero included.

14.6.4 EXTBIO.INC

The *extended BIOS* is how a program finds routines offered by a cartridge, such as MSX-DOS2's own mapper routines or MSX-MIDI. The names start with `EXTBIO_` and the name of the device, as in `EXTBIO_DOS2_DEVICE`, because the names the manuals give, such as `INIT` or `EOF`, would clash with names in almost any program. The comment at the start of the file explains how the extended BIOS is called.

14.6.5 ASCII.INC

The control codes, from `CHR_NUL` to `CHR_DEL`, and what the MSX does with each. Their values are decimal, unlike those of the other files, because character codes are usually written in decimal, as in `db 13,10`: `CHR_CR` is 13 and `CHR_LF` is 10.

CHAPTER 15

Conditional assembly

15.1 IF, ELSE and ENDIF 103
15.2 The two passes 105
15.3 IFDEF and IFNDEF 106
15.4 IF1 and IF2 107
15.5 IFB, IFNB, IFIDN and IFDIF 107
15.6 Nesting 108
15.7 Messages 108

Conditional assembly lets a source assemble some of its lines and skip others, depending on a condition that TATARA works out while it reads the source. One source can then make more than one program: a version with extra checks for testing and one without, or a version for the MSX1 and one for the MSX2, chosen by changing one line.

This chapter describes the conditional directives, and the two passes that TATARA makes over the source, which every condition depends on.

15.1 IF, ELSE and ENDIF

`IF` is followed by an expression. If its value is not zero, the lines after it are assembled; if it is zero, they are skipped. `ENDIF` ends the lines that `IF` controls, and `ELSE`, which is optional, starts the lines to assemble instead when the condition is false. `IFE` is the opposite test: its lines are assembled when the value is zero.

```
debug           equ     1
                cseg
                if      debug
                db      'D'
                else
                db      'R'
                endif
                ife     debug
                db      'E'
                endif
```

Here `db 'D'` is assembled, and `db 'R'` and `db 'E'` are skipped; changing the first line to `debug equ 0` swaps them. The listing shows every line, but the skipped ones have no address and no bytes:

MSX-DOS

```
  0001                          debug           equ     1
                                                cseg
                                                if      debug
  0000'   44                                    db      'D'
                                                else
                                                db      'R'
                                                endif
                                                ife     debug
                                                db      'E'
                                                endif
```

Chapter 18 shows how to leave skipped lines out of the listing altogether.

Any value other than zero counts as true. The comparison operators of chapter 8 give 0FFFFh for true and 0 for false, so they can be used directly: `if size gt 256`.

Some of these directives have other names, which TATARA accepts so that sources written for M80 assemble unchanged (table 15.1).

Other name	Same as
`IFT`, `COND`	`IF`
`IFF`	`IFE`

Table 15.1: Other names of the conditional directives.

M80 also accepts `ENDC` for `ENDIF`. TATARA does not; in a source written for M80, change it to `ENDIF`.

15.2 The two passes

A label can be used before the line that defines it, as in the first line here:

```
                jp      later
later:          ret
```

When TATARA reaches `jp later`, it has not seen `later` yet, so it cannot know its address. It gets round this by reading the source twice. Each reading is called a *pass*.

- On the *first pass*, TATARA works out how many bytes each line takes, and so the address of every label and the value of every name. It writes nothing.
- On the *second pass*, every value is known, and TATARA produces the bytes of the program, the listing and the object file.

The first panel of figure 15.1 shows the two passes over those two lines. Almost every assembler works this way, M80 included.

Source			**First pass**	**Second pass**
	`jp`	`later`	0000: 3 bytes, `later` unknown	`0000: C3 03 00`
`later:`	`ret`		`later` = 0003	`0003: C9`

Source			**First pass**	**Second pass**
	`ifdef`	`later`	not defined yet: skip	defined: assemble
	`db`	`1`	(skipped)	`0000: 01`
	`endif`			
`after:`	`nop`		`after` = 0000	**`after` = 0001: phase error**
`later`	`equ`	`5`	`later` = 5	`later` = 5

Figure 15.1: What each pass does. Above, a label used before its line. Below, a condition that gives a different answer on each pass.

The names and values found on the first pass are kept for the second. So on the second pass, every name that the first pass defined is already defined, including names whose lines come later in the source.

This matters for conditional assembly, because TATARA evaluates each condition on both passes. If a condition gives a different answer on the second pass, the second pass assembles different lines from the first, and the labels after them

move. The second panel of figure 15.1 shows it: `IFDEF` asks whether `later` is defined (section 15.3). On the first pass it is not yet, so `db 1` is skipped and `after` is 0000h. On the second pass it is, so `db 1` is assembled and `after` would be 0001h. TATARA stops at the line of `after`:

MSX-DOS

```
DEFLATE.AS(6): ERROR: phase error - this label had a different value on pass 1.
```

If a condition makes the second pass skip a label that the first pass defined, the message is the one in section 13.6 instead.

So a condition must give the same answer on both passes. The simplest way to be sure is to use only names that are defined above the condition. For `IF`, TATARA requires it: a name defined later stops it on the first pass.

```
; FWD.AS - IF of a name defined later.
                cseg
                if      later
                db      1
                endif
later           equ     1
```

MSX-DOS

```
FWD.AS(3): ERROR: undefined symbol in an expression.
```

15.3 IFDEF and IFNDEF

`IFDEF` is followed by a name, and its lines are assembled if the name is defined; `IFNDEF` assembles them if it is not. A name declared with `EXTRN` counts as defined.

A common use is a value that a source may set for itself, with a default otherwise:

```
                ifndef  bufsize
bufsize         equ     128
                endif
```

A source that wants another size defines `bufsize` before these lines. It must be *before* them: as section 15.2 showed, a name defined further down is undefined on the first pass and defined on the second, and the answer changes between the two.

For the same reason, `IFNDEF` cannot keep an include file from being read twice (section 13.6).

15.4 IF1 and IF2

`IF1` assembles its lines on the first pass only, and `IF2` on the second pass only.

They come from M80, and are kept so that M80 sources assemble. They were used for things that should happen once, not twice, in the days when a listing went straight to a printer. In TATARA they are of little use. The listing shows the second pass, so it shows `IF2`'s lines and not `IF1`'s. And anything that defines a label or produces bytes inside `IF1` or `IF2` makes the two passes different, which section 15.2 forbids:

```
; PASSLAB.AS - a label inside IF1.
                cseg
                if1
here:           nop
                endif
```

MSX-DOS

```
PASSLAB.AS(4): ERROR: this label was defined on pass 1 and not on pass 2 - a
    conditional skipped it.
```

15.5 IFB, IFNB, IFIDN and IFDIF

These four test text, not values:

- `IFB` assembles its lines if the text is blank, and `IFNB` if it is not.
- `IFIDN` assembles its lines if two pieces of text are identical, and `IFDIF` if they differ. The two are separated by a comma.

Each piece of text is written between angle brackets, which are not part of it. For a piece that holds no spaces or commas, the brackets can be left out:

```
                ifb     <>              ; true: blank
                ifb     <x>             ; false
                ifnb    <x>             ; true
                ifdif   <abc>,<abd>     ; true
                ifidn   <abc>,<ABC>     ; false: the case differs
                ifidn   abc,abc         ; true
```

The comparison is exact, upper and lower case included, as in M80, and `/C` makes no difference to it.

With text written out like this, the answer is always the same, so these directives are of no use on their own. They are used inside macros, where the text is an argument that changes from one use of the macro to the next (chapter 16).

15.6 Nesting

A conditional can be inside another. An `IF` inside lines that are being skipped is not even evaluated, and none of its lines are assembled, whatever its condition:

```
                if      0
                if      1
                db      1               ; skipped: the outer IF is false
                else
                db      2               ; skipped too
                endif
                else
                if      1
                db      3               ; assembled
                else
                db      4
                endif
                endif
```

Each `ELSE` and `ENDIF` belongs to the innermost `IF` that is still open. TATARA allows up to 16 conditionals open at once.

A label can be written on an `IF`, `ELSE` or `ENDIF` line, as on any other. It is defined, with the address of the location counter, if TATARA is assembling when it reaches the line, and not defined if it is skipping, as M80 does. So a label on the `ELSE` that follows skipped lines is not defined, and neither is one on an `IF` inside skipped lines. The safest place for a label is a line of its own.

15.7 Messages

Table 15.2 lists the messages about conditional assembly, each with the line that causes it.

When a conditional is never closed, TATARA notices only at the end of the source, but the message names the line of the `IF`, so that it can be found. If several are open, it names the outermost one.

Line	Message
`ELSE` or `ENDIF` with no `IF` open, or a second `ELSE` for one `IF`	`ELSE or ENDIF without a matching IF.`
an `IF` never closed	`conditional not closed by ENDIF.`
a seventeenth `IF` inside sixteen others	`conditionals nested too deeply.`
`if` with no expression	`bad expression.`
`if later`, `later` defined further down	`undefined symbol in an expression.`
`endc`	`not a directive, a macro or an instruction.`

Table 15.2: Messages about conditional assembly.

CHAPTER 16

Macros

16.1 Defining and using a macro 112
16.2 Arguments 113
16.3 Joining text with & 114
16.4 Comments in a body 115
16.5 LOCAL 115
16.6 EXITM 116
16.7 Macros inside macros 117
16.8 Upper and lower case 118
16.9 Messages 118

A *macro* is a name for a group of lines. Once the group is defined, the name can be used like an instruction, and TATARA replaces each use with the lines of the group. The result of one use is called an *expansion*.

The `system` macro of chapter 14 is one: `system _STROUT` is replaced by `ld c,_STROUT` and `call BDOS`. A macro is not a subroutine. A subroutine exists once in the program and is reached with `call`; a macro's lines are put into the program again at every use. A macro saves typing, not bytes.

This chapter describes how to define a macro, how to pass it values, and the directives that are used inside one. `REPT`, `IRP` and `IRPC`, which repeat lines without a name, are in chapter 17; they follow the same rules for arguments, `LOCAL` and `EXITM`.

16.1 Defining and using a macro

A macro is defined with `MACRO`, which is preceded by the name of the macro and followed by the names of its *parameters*, separated by commas. The lines that follow are its *body*, and `ENDM` ends it:

```
fill            macro   addr,value
                ld      hl,addr
                ld      (hl),value
                endm
```

As with `EQU`, the name is written without a colon (chapter 12). The definition produces no bytes; TATARA only stores the body. A macro must be defined before its first use.

To use the macro, write its name as if it were an instruction, followed by the *arguments*: the values its parameters stand for in this use. TATARA replaces each parameter in the body with the matching argument, the first argument for the first parameter, the second for the second (figure 16.1).

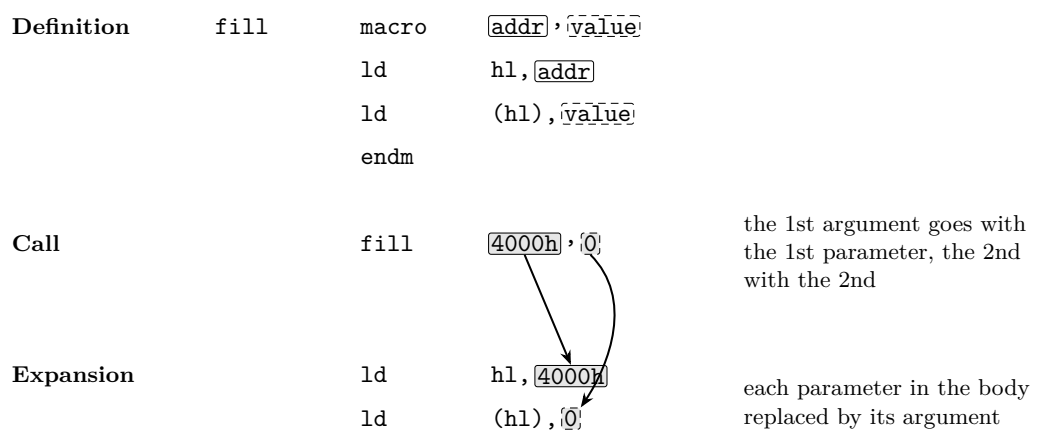


Figure 16.1: A macro's definition, a use of it, and the expansion.

The listing shows each use followed by its expansion, with the generated lines marked `+`:

MSX-DOS

```
                                                fill    4000h,0
0000'   21 00 40        +                      ld      hl,4000h
0003'   36 00           +                      ld      (hl),0
                                                fill    4001h,0ffh
0005'   21 01 40        +                      ld      hl,4001h
0008'   36 FF           +                      ld      (hl),0ffh
```

Chapter 18 describes how to show less of an expansion in the listing, or none of it.

16.2 Arguments

The arguments of a use are separated by commas. Blanks before an argument are not part of it.

- An argument that is missing is empty. The parameter is then replaced by nothing, and the body has to allow for that: with `db 0,b` in the body, an empty `b` leaves `db 0,`, which TATARA refuses. `IFB` and `IFNB` (section 16.6) test for it.
- Arguments beyond the last parameter are ignored, as in M80.
- An argument that holds a comma is written between angle brackets, which are removed: `<7,8>` is one argument, `7,8`.
- `!` before a character makes it part of the argument whatever it is, such as a `>` or a comma that would otherwise end the argument, or a semicolon that would otherwise start a comment. A semicolon between angle brackets is part of the argument too.

This macro puts its first argument in a `DB`, and its second in another if there is one:

```
show            macro   a,b
                db      a
                ifnb    <b>
                db      b
                endif
                endm
```

In these uses, `show 3` leaves `b` empty, the `6` in `show 4,5,6` is ignored, `<7,8>` is one argument, and `'!>'` passes a string holding `>`:

```
                                        show    3
0002'   03              +               db      3
                                        show    4,5,6
0003'   04              +               db      4
0004'   05              +               db      5
                                        show    <7,8>,9
0005'   07 08           +               db      7,8
0007'   09              +               db      9
                                        show    '!>',10
0008'   3E              +               db      '>'
0009'   0A              +               db      10
```

An argument is text, not a value: `five*2` is passed as the six characters `five*2`. A `%` in front of an argument passes its value instead, written as a decimal number. With a body of `db '&x'` (the `&` is explained in the next section), the first use below passes the text and the second its value:

MSX-DOS

```
                                        show    five*2
0000'   66 69 76 65     +               db      'five*2'
0004'   2A 32           +
                                        show    %five*2
0006'   31 30           +               db      '10'
```

The value must be one TATARA already knows, as for `IF` (chapter 15).

16.3 Joining text with &

A parameter is replaced only where its name stands as a word of its own. To join it to other text, put `&` between them: `lab&n` with the argument `1` becomes `lab1`. The `&` itself disappears.

Inside a string, a parameter is not replaced at all unless it is joined with `&`. In the string `'n'`, the `n` is just a letter; in `'&n'`, it is the parameter:

```
label           macro   n
lab&n:          db      'n'
                db      '&n'
                endm
```

MSX-DOS

```
                                        label   1
0000'   6E              +     lab1:     db      'n'
0001'   31              +               db      '1'
```

16.4 Comments in a body

A comment that starts with `;` is kept in the body and appears in every expansion. One that starts with `;;` is dropped when the macro is defined, so it is seen in the definition only. Use `;;` for notes about how the macro works, which would only repeat themselves in the listing:

```
note            macro
                nop             ; kept
                nop             ;; dropped
                endm
```

MSX-DOS

```
                                        note
0000'   00              +               nop             ; kept
0001'   00              +               nop
```

16.5 LOCAL

A macro that defines a label can be used only once, because a second use defines the label again (chapter 12):

```
wait            macro   n
                ld      b,n
loop:           djnz    loop
                endm
```

MSX-DOS

```
LOCALNO.AS(4): ERROR: this name already has a value.
    in wait, called from LOCALNO.AS(8)
```

`LOCAL`, followed by one or more names, solves this. In each expansion, TATARA replaces those names with new ones, `??0000`, `??0001` and so on, a different one each time:

```
wait            macro   n
                local   loop
                ld      b,n
loop:           djnz    loop
                endm
```

MSX-DOS

```
                                                wait    10
 0000'   06 0A           +                      ld      b,10
 0002'   10 FE           +      ??0000:         djnz    ??0000
                                                wait    20
 0004'   06 14           +                      ld      b,20
 0006'   10 FE           +      ??0001:         djnz    ??0001
```

The generated names are ordinary labels, and appear in the symbol table. Do not use names that start with `??` for labels of your own. `LOCAL` lines must come first in the body, before any other line; otherwise TATARA stops with `LOCAL must come before the macro body.`

16.6 EXITM

`EXITM` ends the expansion at once: the lines of the body after it are not used. It is used inside a conditional, so that the macro stops early in some cases. This macro does nothing when it is given no argument:

```
put             macro   x
                ifb     <x>
                exitm
                endif
                db      x
                endm
```

MSX-DOS

```
                                                put     1
 0000'   01              +                      db      1
                                                put
                                                put     3
 0001'   03              +                      db      3
```

This is where the text tests of chapter 15 are useful. `IFB` and `IFNB` ask whether an argument was given, and `IFIDN` and `IFDIF` compare an argument with a fixed text, such as a register name. The conditionals that `EXITM` leaves open inside the macro are closed with it.

16.7 Macros inside macros

A macro's body can use another macro, and can define one:

```
byte            macro   x
                db      x
                endm
pair            macro   x,y
                byte    x
                byte    y
                endm
maker           macro   name,val
name            macro
                db      val
                endm
                endm
```

Here `pair 1,2` produces `db 1` and `db 2` through `byte`. `maker three,3` produces no bytes, but defines a macro called `three`, and `three` then produces `db 3`. A `MACRO` line in a body is only text until the body is expanded, and each `ENDM` belongs to the innermost `MACRO`.

A macro can also use itself. It then needs a condition that stops it, as in this macro, which counts down to zero:

```
down            macro   n
                db      n
                if      n gt 0
                down    %n-1
                endif
                endm
```

MSX-DOS

```
                                                down    3
 0000'   03              +                      db      3
 0001'   02              +                      db      2
 0002'   01              +                      db      1
 0003'   00              +                      db      0
```

Expansions inside expansions, together with include files, can be 16 deep. Without a condition to stop it, a macro that uses itself reaches that limit, and TATARA stops with `sources nested too deeply.`

A macro can be defined again, and the new definition replaces the old one from that line on. The `system` macro depends on this: each module that includes `MSXDOS.INC` defines it again (chapter 14).

16.8 Upper and lower case

Macro names follow the same rule as other names: without `/C`, `PUT` uses the macro `put`; with `/C`, it does not, and TATARA stops with `not a directive, a macro or an instruction.` (chapter 7).

Parameter names are matched ignoring case in both modes, as in M80: a parameter `val` is replaced wherever the body says `val`, `VAL` or `Val`.

16.9 Messages

Table 16.1 lists the messages about macros, each with the line that causes it.

Line	Message
`macro` with no name before it	`MACRO without a name.`
a macro name of 65 characters	`a macro name may be at most 64 characters.`
`put: macro`	`EQU, DEFL and MACRO take a name, not a label.`
a macro with 17 parameters	`too many macro parameters.`
`bad macro a,,b`	`bad macro parameter list.`
a `MACRO` with no `ENDM`	`macro definition not closed by ENDM.`
`ENDM` with no `MACRO`	`ENDM without a macro definition.`
`LOCAL` after a line of the body	`LOCAL must come before the macro body.`
`EXITM` outside a macro	`EXITM outside a macro or repeat block.`
a macro that uses itself with no end	`sources nested too deeply.`

Table 16.1: Messages about macros.

When a line in an expansion causes an error, the message names the line of the body, in the file where the macro was defined, and the line below it says which macro it was and where it was used:

```
ERRIN.AS(4): ERROR: not a form this instruction has.
    in bad, called from ERRIN.AS(8)
```

Chapter 20 describes these trails in full.

CHAPTER 17

Repeat blocks

17.1 REPT 121
17.2 IRP 122
17.3 IRPC 124
17.4 LOCAL, EXITM and nesting 124
17.5 Messages 125

A *repeat block* is a group of lines that TATARA repeats where it stands. It works like a macro without a name, which is defined and used in the same place. `REPT` repeats its lines a given number of times, `IRP` once for each item in a list, and `IRPC` once for each character in a piece of text.

All three end with `ENDM`, as a macro does, and follow the rules of chapter 16 for their body, for `LOCAL` and for `EXITM`.

17.1 REPT

`REPT` is followed by a count, and the lines up to `ENDM` are assembled that many times:

```
                cseg
here:           rept    3
                nop
                endm
                dw      here
```

MSX-DOS

```
                                here:           rept    3
                                                nop
                                                endm
0000'   00              +                       nop
0001'   00              +                       nop
0002'   00              +                       nop
0003'   00 00                                   dw      here
```

The listing shows the block as it was written, then each round, marked + as in a macro expansion. A label on the `REPT` line names the start of the first round: `here` is 0000h.

The count can be any expression, but TATARA works it out before it reads the lines of the block, so it must be a number TATARA already knows: a name defined further down stops TATARA with `undefined symbol in an expression.`, and a relocatable value, such as a label in a code segment, with the relocation error of chapter 8. A count of 0 assembles nothing.

A repeat block is where `DEFL` (chapter 12) is most useful. A name defined with `DEFL` can be given a new value in each round, and so can count the rounds:

```
n               defl    0
                rept    4
                db      n
n               defl    n+1
                endm
```

MSX-DOS

```
0000'   00              +                       db      n
0001'   01              +                       db      n
0002'   02              +                       db      n
0003'   03              +                       db      n
```

The listing shows each line as it was written, `db n`, and the bytes show the value `n` had in that round.

17.2 IRP

`IRP` is followed by a parameter name, a comma, and a list of items between angle brackets. The block is assembled once for each item, with the parameter replaced by that item:

```
                irp     x,<1,2,4,8>
                db      x
                endm
```

MSX-DOS

```
0000'   01              +                       db      1
0001'   02              +                       db      2
0002'   04              +                       db      4
0003'   08              +                       db      8
```

The items follow the rules of a macro's arguments (section 16.2). An item that holds a comma is written between angle brackets of its own, and % passes a value instead of text:

```
five            equ     5
                irp     s,<'ab',<'c,d'>>
                db      s
                endm
                irp     v,<five*2,%five*2>
                db      '&v'
                endm
```

MSX-DOS

```
0005'   61 62           +                       db      'ab'
0007'   63 2C 64        +                       db      'c,d'
000A'   66 69 76 65     +                       db      'five*2'
000E'   2A 32           +
0010'   31 30           +                       db      '10'
```

The angle brackets round the list can be left out: `irp x,1,2,3` is the same as `irp x,<1,2,3>`. An empty list, `irp x,<>`, gives one round, with the parameter empty.

`IRP` is most useful inside a macro, over an argument that is itself a list:

```
bytes           macro   list
                irp     b,<list>
                db      b
                endm
                endm
```

MSX-DOS

```
                                                bytes   <10,20,30>
0000'   0A              +                       db      10
0001'   14              +                       db      20
0002'   1E              +                       db      30
```

17.3 IRPC

`IRPC` is followed by a parameter name, a comma, and a piece of text. The block is assembled once for each character of the text, with the parameter replaced by that character. As in a macro, a parameter inside a string needs `&` (section 16.3):

```
                irpc    c,TATARA
                db      '&c'
                endm
```

MSX-DOS

```
0000'   54              +                       db      'T'
0001'   41              +                       db      'A'
0002'   54              +                       db      'T'
0003'   41              +                       db      'A'
0004'   52              +                       db      'R'
0005'   41              +                       db      'A'
```

Every character counts, spaces and commas included: `irpc c,<A B,C>` gives five rounds, for `A`, a space, `B`, a comma and `C`. The angle brackets are needed only to keep a space in the text, and are not part of it. An empty text, `irpc c,<>`, gives no rounds at all. `!` and `%` have no special meaning in the text of `IRPC`.

17.4 LOCAL, EXITM and nesting

`LOCAL` works in a repeat block as in a macro, and gives each round names of its own:

```
                rept    2
                local   skip
                jr      skip
                nop
skip:
                endm
```

MSX-DOS

```
0000'   18 01           +                       jr      ??0000
0002'   00              +                       nop
0003'   18 01           +                       jr      ??0001
0005'   00              +                       nop
```

Without `LOCAL`, the second round would define `skip` again. A line that holds only a label produces no bytes, and is not shown in the expansion.

`EXITM` ends the whole block, including the rounds that have not run yet. This block would run ten times, but stops after three:

```
n               defl    0
                rept    10
                if      n eq 3
                exitm
                endif
                db      n
n               defl    n+1
                endm
                db      0ffh
```

MSX-DOS

```
0000'   00              +                       db      n
0001'   01              +                       db      n
0002'   02              +                       db      n
0003'   FF                                      db      0ffh
```

Repeat blocks can be put inside each other and inside macros, and each `ENDM` ends the innermost one. Together with macro expansions and include files, they can be 16 deep (section 16.7).

17.5 Messages

Table 17.1 lists the messages about repeat blocks, each with the line that causes it. When an error happens inside a repeat block, the trail that follows the message names it as `a repeat block`.

Line	Message
`rept count`, `count` defined further down	`undefined symbol in an expression.`
`rept start`, `start` a label in a code segment	`relocation error - a segment-relative value is not allowed here.`
`irp x`, with no comma	`bad macro parameter list.`
a repeat block with no `ENDM`	`macro definition not closed by ENDM.`

Table 17.1: Messages about repeat blocks.

CHAPTER 18

The listing

18.1 A listing line 127
18.2 Pages 129
18.3 Turning the listing off and on 132
18.4 Macro expansions 132
18.5 Lines not assembled 134
18.6 Messages 135

A listing shows what TATARA made of the source: each line as it was written, with the address it was assembled at and the bytes it produced. It is where to look when a program does not do what the source seems to say, or when a line takes more room than expected.

Chapter 5 describes how to ask for one. A listing goes to the file named third on the command line or, when there is none, to the screen with `/P`. `/L` puts the address and the bytes in front of each line; without it, a listing holds only the source. This chapter describes listings made with `/L`. To see one on the screen instead of in a file, give both options:

```
tatara /l /p modes.as
```

The rest of the chapter describes what a listing line holds, how the listing is divided into pages, and the directives with which a source decides what the listing shows.

18.1 A listing line

Figure 18.1 shows the columns of a listing line. The address comes first, then the bytes, then the source line, unchanged.

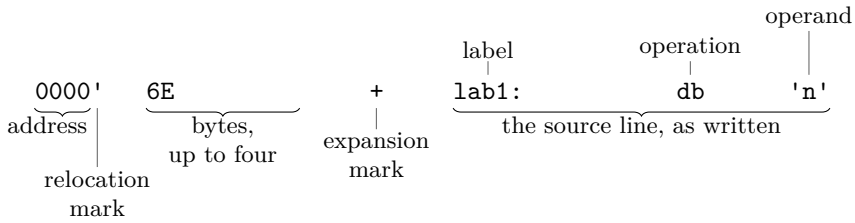


Figure 18.1: The columns of a listing line.

This source uses one example of each kind of line:

```
                extrn   far
five            equ     5
                aseg
                org     4000h
abs:            nop
                cseg
code:           ld      hl,data
                call    far
                db      1,2,3,4,5,6,7,8,9
only:
                dseg
data:           ds      2
                end
```

MSX-DOS

```
                                        extrn   far
 0005                       five        equ     5
                                        aseg
                                        org     4000h
 4000   00                  abs:        nop
                                        cseg
 0000'  21 00 00            code:       ld      hl,data
 0003'  CD 00 00                        call    far
 0006'  01 02 03 04                     db      1,2,3,4,5,6,7,8,9
 000A'  05 06 07 08
 000E'  09
 000F'                      only:
                                        dseg
 0000'                      data:       ds      2
                                        end
```

- **The address** is where the line was assembled, in hexadecimal. A line that produces no bytes but defines a label, such as `only:` and `data: ds 2`, shows the address the label was given. On an `EQU` line, the address column shows the value of the name instead.

- **The relocation mark**, an apostrophe, follows an address that is relocatable: an offset in the code or the data segment, which TANREN finishes (chapter 10). An address in the absolute segment, such as 4000h, has no mark. The mark is the same for code and data.
- **The bytes** are the ones the line produced, in the order they are in memory, up to four on a line. A line that produces more continues on the lines below it, which show only the address and the bytes: the `DB` of nine bytes takes three lines.
- **The source line** follows, as it was written.

Lines that produce nothing and define nothing, such as `EXTRN`, `ASEG` and `ORG`, have the address and bytes columns empty.

When an operand is relocatable or external, the bytes are the ones the object file holds before TANREN finishes them: the offset of `data` in its segment, 0000h, and zeros for the external name `far`. The listing does not mark these bytes. M80's listing shows such an operand as one 16-bit value followed by a mark, `0000"` or `0000*`; TATARA shows each byte, in memory order, and no mark.

A line that comes from a macro expansion or a repeat block has a `+` in the column before the source line (chapter 16).

18.2 Pages

A listing is divided into pages. Each page starts with a *page heading*: a line with the title, the name and version of TATARA, the date and the page number, then a line for the subtitle and a blank line. Every page after the first starts with a form feed, so that a printer starts a new sheet. By default a page holds 56 lines of the source.

Three directives control the pages:

- `TITLE text` sets the title, which appears at the left of the page heading.
- `SUBTTL text` sets the subtitle, which appears on the line under the page heading.
- `PAGE` starts a new page after its own line. `PAGE n` sets the length of a page to `n` lines and starts a new page too. The count is the whole page, four lines of which go to the page heading, so `page 12` leaves eight lines for the source. `n` must be from 10 to 255. M80's `*EJECT` and `$EJECT`, written in column 1, are other names for `PAGE` without a count.

TATARA keeps the first 28 characters of a title or a subtitle, and drops the rest.

This source sets a title and a subtitle, makes the pages short, and starts a new page for its second part:

```
                title   Page test
                subttl  First part
                page    12
                cseg
                db      1
                db      2
                ...
                db      10
                subttl  Second part
                page
                db      11
                end
```

MSX-DOS

```
Page test       Tatara v1.2.0   30-Sep-26       PAGE    1


                                                title   Page test
                                                subttl  First part
                                                page    12
Page test     Tatara v1.2.0   30-Sep-26       PAGE    1-1
First part

                                                cseg
  0000'   01                                    db      1
  0001'   02                                    db      2
  0002'   03                                    db      3
  0003'   04                                    db      4
  0004'   05                                    db      5
  0005'   06                                    db      6
  0006'   07                                    db      7
Page test     Tatara v1.2.0   30-Sep-26       PAGE    1-2
First part

  0007'   08                                    db      8
  0008'   09                                    db      9
  0009'   0A                                    db      10
                                                subttl  Second part
                                                page
Page test     Tatara v1.2.0   30-Sep-26       PAGE    1-3
Second part

  000A'   0B                                    db      11
                                                end
```

The headings after the first sit two columns further left because each starts with the form feed, which takes no room on the page. The listing shows these points:

- The first page already has the title, although the `TITLE` line comes after its heading. TATARA keeps the title from the first pass (chapter 15), so the first page shows the last title in the source. After that, each page shows the title in force when it starts. A source with one `TITLE`, at the top, has it on every page.
- A subtitle is not kept from the first pass, and appears only from the page after the `SUBTTL` line. To start a new part of the source on a new page with its own subtitle, write `SUBTTL` and then `PAGE`, as above.
- The page numbers go 1, 1-1, 1-2, 1-3. A page started by `PAGE`, or by the previous page filling up, is numbered as a part of the same page: its main number stays and the number after the dash goes up.

The main number goes up at a form feed in the source. A line holding only a form feed (character 0Ch) starts a new page, numbered 2, 3 and so on, and the pages after it are 2-1, 2-2 and so on. This is how M80 numbers pages, and it lets a source divided by form feeds be printed with one main number for each part.

18.2.1 The last page

A listing written to a file ends with a page numbered `S`, which lists the names of the macros the source defined and then the symbols, with their values:

MSX-DOS

```
        Tatara v1.2.0   30-Sep-26       PAGE    S


Macros:

Symbols:
000F'   only            0000*   far             0000'   data
4000    abs             0000'   code            0005    five
```

This is the listing of the source in section 18.1, which defines no macros. A relocatable value has the apostrophe it has in a listing line. An external name has no value in this module, and is shown as `0000*`. The symbol table is described in chapter 19. A listing printed on the screen with `/P` does not have this page.

18.3 Turning the listing off and on

`.XLIST` turns the listing off, and `.LIST` turns it back on. The lines between them are assembled as usual, but not listed. `.XLIST` itself is not listed either; `.LIST` is:

```
                cseg
                db      1
                .xlist
                db      2
                db      3
                .list
                db      4
                end
```

MSX-DOS

```
                                                cseg
0000'   01                                      db      1
                                                .list
0003'   04                                      db      4
                                                end
```

The address goes from 0000h to 0003h, because the two lines that are not listed still produced a byte each.

The most common use is around an include file, whose lines are otherwise all in the listing (chapter 5):

```
                .xlist
                include msxdos.inc
                .list
```

18.4 Macro expansions

Three directives decide how much of a macro expansion or a repeat block the listing shows. Each applies from its own line on, until another of the three.

- `.XALL`, the default, lists the lines of an expansion that produce bytes.
- `.LALL` lists every line of an expansion, including the lines of a conditional that are not assembled.
- `.SALL` lists no line of an expansion. The line that uses the macro is listed with a `+`, to show that it stands for an expansion.

This macro has a line that is never assembled. The source uses it once in each mode:

```
two             macro
                db      1
                if      0
                db      2
                endif
                ld      a,3
                endm
                cseg
                two
                .lall
                two
                .sall
                two
                .xall
                two
                end
```

MSX-DOS

```
                                        two
0000'   01              +               db      1
0001'   3E 03           +               ld      a,3
                                        .lall
                                        two
0003'   01              +               db      1
                        +               if      0
                        +               db      2
                        +               endif
0004'   3E 03           +               ld      a,3
                                        .sall
                        +               two
                                        .xall
                                        two
0009'   01              +               db      1
000A'   3E 03           +               ld      a,3
```

Under `.LALL`, the lines that were not assembled have no address. Under `.SALL`, the only signs of the expansion are the + on the line that uses the macro and the addresses, which go from 0004h to 0009h: the expansion's three bytes are at 0006h to 0008h.

18.5 Lines not assembled

Three more directives decide whether the listing shows the lines that a conditional (chapter 15) leaves out:

- `.LFCOND`, the default, lists them, with no address.
- `.SFCOND` leaves them out. The `IF`, `ELSE` and `ENDIF` lines are still listed.
- `.TFCOND` changes from one of these to the other: after `.LFCOND` it works as `.SFCOND`, and after `.SFCOND` as `.LFCOND`.

This source has five conditionals that are false, one before each directive and one after the last:

```
                cseg
                if      0
                db      1
                endif
                .sfcond
                if      0
                db      2
                endif
                .lfcond
                if      0
                db      3
                endif
                .tfcond
                if      0
                db      4
                endif
                .tfcond
                if      0
                db      5
                endif
                end
```

MSX-DOS

```
                                        cseg
                                        if      0
                                        db      1
                                        endif
                                        .sfcond
                                        if      0
                                        endif
                                        .lfcond
                                        if      0
                                        db      3
```

```
                                                endif
                                                .tfcond
                                                if      0
                                                endif
                                                .tfcond
                                                if      0
                                                db      5
                                                endif
                                                end
```

The lines of the second and fourth conditionals are not listed. M80 lists this source in the same way.

18.6 Messages

A `PAGE` length outside 10 to 255 stops TATARA with this message:

MSX-DOS

```
PAGEBAD.AS(1): ERROR: a PAGE length must be 10 to 255.
```

The other listing directives take no operand, or take any text, and have no messages of their own.

CHAPTER 19

The symbol table dump and the diagnostic switches

19.1 The symbol table 137
19.2 Reading the table 139
19.3 The table and the listing 142
19.4 Diagnostic switches 142

`/S` makes TATARA print the *symbol table* when the assembly ends: every name the source defined, with its value, grouped by segment. It is the quickest way to check where a label ended up, what value an `EQU` was given, or which names a module makes public and which it expects from others.

The chapter also describes `/F`, `/M` and `/H`, the three options that show how TATARA read a source. They are not needed to write programs, but their output is what to send with a report of a problem in TATARA.

19.1 The symbol table

`/S` prints the table after the last line has been assembled and before the summary line, so that the summary line is still the last thing on the screen. The table always goes to the screen, even when a listing file is written.

This source defines a name of each kind:

```
                public  start,count,limit,level
                extrn   print,exit
limit           equ     100
level           defl    1
level           defl    2
tmp             defl    3
                cseg
start:          ld      hl,msg
                call    print
                call    MIXED
                jp      exit
wait            macro
                local   loop
loop:           djnz    loop
                endm
                wait
                wait
Mixed:          ret
after           equ     start+1
                dseg
count:          ds      1
msg:            db      'Hi$'
                cseg    music
tune:           db      0
                end
```

MSX-DOS

```
A:\>tatara /s syms.as syms.tro
Tatara MSX Macro-Assembler v1.2.0
Copyright (C) 2026 Javier Lavandeira
https://tatara.tools

ASEG - absolute

0002h  public var level
0003h  var        tmp
0064h  public     limit

CSEG - default code segment, 0011h bytes

0000h  public     start
0001h             after
000Ch             ??0000
000Eh             ??0001
0010h             Mixed

DSEG - default data segment, 0004h bytes

0000h  public     count
0001h             msg
```

```
music - named code segment, 0001h bytes

0000h             tune

EXTERNAL - resolved by the linker

                  exit
                  print

ended at SYMS.AS(26)
```

With `/Q` as well, the banner and the summary line are left out and only the table is printed.

19.2 Reading the table

19.2.1 Sections

The table has one section for each segment of the module (chapter 10), then one for the external names, then one for the names declared `PUBLIC` and never defined. Each section starts with a heading line; table 19.1 lists the forms it takes.

The size in a heading is the number of bytes the segment takes in this module. The absolute segment has no size, because its bytes can be anywhere in memory.

The three default segments always have a section, even when they have no symbols. A source that defines nothing still prints these:

MSX-DOS

```
ASEG - absolute

(no symbols)

CSEG - default code segment, 0001h bytes

(no symbols)

DSEG - default data segment, 0000h bytes

(no symbols)
```

The sections for named segments, external names and undefined names appear only when there is something to put in them.

Heading	Section
`ASEG - absolute`	the absolute segment, and every absolute value
`CSEG - default code segment, 0011h bytes`	the default code segment
`DSEG - default data segment, 0004h bytes`	the default data segment
`music - named code segment, 0001h bytes`	a named code segment
`scratch - named data segment, transient, group reading, 0042h bytes`	a named data segment; transient and with a group (section 10.6)
`EXTERNAL - resolved by the linker`	names declared `EXTRN`
`UNDEFINED - named by PUBLIC, never defined`	names declared `PUBLIC` and not defined

Table 19.1: Section headings in the symbol table.

19.2.2 Symbol lines

Each line in a segment's section has the symbol's value, its type and its name. The lines are in order of value.

- **The value** is an offset in the segment: `Mixed` is 10h bytes from the start of the code segment, wherever TANREN puts it. In the absolute segment it is the value itself. An `EQU` whose value is relocatable, such as `after equ start+1`, belongs to the segment of that value. A `DEFL` name shows the last value it was given: `level` is 2.
- **The type** is one of the words in table 19.2.
- **The name** is spelled as it was written where it was defined. The source uses `MIXED` before it defines `Mixed:`, and the table shows `Mixed`. With `/C`, these would be two different names (chapter 7).

The names `??0000` and `??0001` are the ones `LOCAL` made up for the two uses of `wait` (chapter 16). They are ordinary labels, one for each use.

Type	The name is
(blank)	a label or an `EQU` name
`public`	a label or an `EQU` name made public
`var`	a `DEFL` name
`public var`	a `DEFL` name made public

Table 19.2: The types in the symbol table.

19.2.3 Names with no value

External names have no value in the module that uses them: TANREN finds it in another module (chapter 25). They are listed with no value and no type, in no particular order.

A name declared `PUBLIC` and never defined stops TATARA when it writes an object file (chapter 12). Without an object file, the source is accepted, and `/S` lists the name in a section of its own. This source declares two public names and defines only one:

```
                public  here,nowhere
here:           ret
                end
```

MSX-DOS

```
A:\>tatara /s undef.as
...
CSEG - default code segment, 0001h bytes

0000h  public     here

DSEG - default data segment, 0000h bytes

(no symbols)

UNDEFINED - named by PUBLIC, never defined

                  nowhere
```

19.3 The table and the listing

A listing file ends with a page of symbols of its own (section 18.2.1). For the source in section 19.1, it is this:

MSX-DOS

```
Macros:
wait

Symbols:
000E'   ??0001          000C'   ??0000          0000*   exit
0003    tmp             0000'   tune            0001'   after
0010'   Mixed           0000'   count           0001'   msg
0000'   start           0002    level           0000*   print
0064    limit
```

It holds the same symbols, in M80's layout: three to a line, in no particular order, with `'` after a relocatable value and `*` after an external name. It does not say which segment a value belongs to, or whether a name is public: `tune` and `start` are both at `0000'`, one in `music` and one in the default code segment. The table that `/S` prints says both, and is the one to read when that matters.

The two can be asked for together, with `/L` and `/S`: the page goes into the listing file, and the table to the screen.

19.4 Diagnostic switches

`/F`, `/M` and `/H` show how TATARA read a source, rather than what it made of it. If TATARA does something unexpected with a line, run the source again with the option that concerns it, and send the output with the report. MSX-DOS can save the output in a file with `>`:

```
tatara /f prog.as prog.tro > fields.txt
```

19.4.1 /F: the fields of each line

`/F` prints each line as TATARA divided it into fields (chapter 6), instead of listing it. This source has a comment, a line skipped by a conditional, a macro, an include file and a string that holds a semicolon:

```
; FIELDS.AS - what /F shows.
two             macro   x
                db      x,x
                endm
                cseg
start:          ld      hl,msg          ; a comment
                if      0
                nop
                endif
                two     5
                include finc.inc
msg:            db      'a;b'
                end
```

`FINC.INC` holds a comment line and `ret`.

MSX-DOS

```
A:\>tatara /f fields.as fields.tro
...
[][][][; FIELDS.AS - what /F shows.][00][00:0001]
[two][macro][x][][01][00:0002]
[][cseg][][][1A][00:0005]
[start][ld][hl,msg][; a comment][00][00:0006]
[][if][0][][08][00:0007]
[][nop][][][00][00:0008]
[][endif][][][13][00:0009]
[][two][5][][00][00:000A]
[][db][5,5][][20][00:0003]
[][include][finc.inc][][14][00:000B]
[][][][; FINC.INC - included by FIELDS.AS.][00][01:0001]
[][ret][][][00][01:0002]
[msg][db]['a;b'][][20][00:000C]
[][end][][][15][00:000D]
ended at FIELDS.AS(13)
```

Each line of the output has six parts in square brackets:

1. the label, without its colon;
2. the operation;
3. the operands;
4. the comment;
5. a number that TATARA gives each directive, in hexadecimal, or `00` for a line that is not a directive;

6. where the line came from: the number of the file, `00` for the source named on the command line and `01` and so on for include files in the order they were opened, then the line number in that file, in hexadecimal.

The output shows each line as it is read on the first pass. Lines skipped by a conditional are shown, as `nop` is here. A line from a macro expansion shows the line of the macro's body it came from: `db 5,5` is line 3, where `db x,x` was written. The lines of a macro's body are shown only when it is used, not when it is defined. The semicolon inside `'a;b'` is part of the operand, not the start of a comment.

`/F` prints to the screen, with or without `/P`.

19.4.2 /M: macro definitions

`/M` prints each macro definition as TATARA stored it, when the definition is read:

```
; MDUMP.AS - what /M shows.
put             macro   x,y
                local   skip
                ld      a,x             ; kept
                jr      skip            ;; dropped
lab&y:          db      '&x'
skip:
                endm
                cseg
                put     1,2
                rept    2
                nop
                endm
                end
```

MSX-DOS

```
A:\>tatara /m mdump.as mdump.tro
...
MACRO put: 2 parameters, 1 locals, 4 lines, MDUMP.AS(2)
  4 |           ld      a,<0>           ; kept
  5 |           jr      <L0>
  6 | lab<1>:           db      '<0>'
  7 | <L0>:
ended at MDUMP.AS(14)
```

The first line gives the macro's name, the number of parameters, of `LOCAL` names and of lines in its body, and where the definition starts. Each line of the body follows, with its line number in the source:

- a parameter is shown as `<0>`, `<1>` and so on, in the order of the `MACRO` line,

and a `LOCAL` name as `<L0>`, `<L1>` and so on. `lab&y` has become `lab<1>`: the `&` has done its work (section 16.3);

- the `LOCAL` line and the `ENDM` line are not stored;
- a comment that starts with `;;` has been dropped (section 16.4).

The `REPT` block at the end of the source is not printed: `/M` shows macros only, not repeat blocks (chapter 17).

19.4.3 /H: memory in use

TATARA keeps its symbols and macros in blocks of memory in the memory mapper, which it takes as it needs them. `/H` prints, after the summary line, how many blocks are still in use when the assembly ends:

MSX-DOS

```
A:\>tatara /h heap1.as heap1.tro
...
ended at HEAP1.AS(8)
HEAP: 7 blocks.
```

The figure is never zero: each symbol takes a block, each macro definition takes blocks for its name and its body, and TATARA's own records take a few more. It is useful only when compared with the figure for another source. A macro with a `LOCAL` adds a block each time it is used, because each use defines a new `??` name: the same `wait` used three times instead of once gives `HEAP: 9 blocks.`

CHAPTER 20

Error messages

20.1 What a message says 147
20.2 TATARA stops at the first error 149
20.3 The trail 150
20.4 Errors found on the second pass 153
20.5 Internal errors 153

When TATARA finds a mistake in a source, it prints one *error message* and stops. This chapter describes how to read the message, what TATARA leaves on the disk when it stops, and how to follow the *trail* of lines that comes after a message when the mistake is inside an include file, a macro or a repeat block. Appendix D lists every message, with its cause.

20.1 What a message says

A message names the file and the line of the mistake, and says what is wrong:

```
; LERR.AS - an error half way down a listing.
                cseg
                ld      a,1
                ld      b,2
                ld      (bc),b
                ld      c,3
                end
```

MSX-DOS

```
A:\>tatara /l lerr.as lerr.tro lerr.lst
Tatara MSX Macro-Assembler v1.2.0
Copyright (C) 2026 Javier Lavandeira
https://tatara.tools

LERR.AS(5): ERROR: not a form this instruction has.
```

- **The file** is the one that holds the line: the source named on the command line, an include file, or the file where a macro was defined. TATARA prints the name in capitals.
- **The line** is counted from 1 at the top of that file. The comment at the top of `LERR.AS` is line 1, so `ld (bc),b` is line 5.
- **The text** after `ERROR:` says what is wrong. Here `ld (bc),b` is written correctly, but the Z80 has no such instruction.

A definition or a conditional that is never closed is found only at the end of the source. Its message names the line that opened it, the `MACRO` or the `IF`, not the last line of the file.

Some messages have no line to name. A mistake on the command line is found before any file is read, and its message comes before the banner (section 5.6):

MSX-DOS

```
A:\>tatara /z good.as
ERROR: unknown option.
```

A file that cannot be opened is named in the message itself, as in `ERROR: cannot open NOTHERE.INC` (chapter 13).

`/Q` leaves out the banner and the summary line, but never an error message:

MSX-DOS

```
A:\>tatara /q lerr.as
LERR.AS(5): ERROR: not a form this instruction has.
```

20.2 TATARA stops at the first error

TATARA prints one message at a time. When it finds a mistake, it stops there, and does not look at the rest of the source. Correct the line, assemble again, and TATARA goes on to the next mistake, if there is one.

TATARA never leaves a half-written object file. What it leaves depends on when it found the mistake:

- **On the first pass**, where almost every mistake is found, TATARA has not yet created the object file. An object file of the same name from an earlier run is left as it was (chapter 5).
- **On the second pass**, TATARA has already created the new object file. It deletes it, so that no object file of that name is left, not even the one from an earlier run. Here `GOOD.AS` is a correct source, and `PUB.AS` has a name declared `PUBLIC` and never defined, which is found when the object file is finished:

MSX-DOS

```
A:\>tatara good.as pub.tro
...
ended at GOOD.AS(5)
A:\>dir pub.tro
...
PUB      TRO        46 26-09-30  9:04
 46 bytes in 1 file   3800K free
A:\>tatara pub.as pub.tro
...
PUB.AS(7): ERROR: a PUBLIC name was never defined.
A:\>dir pub.tro
...
 0 bytes in 0 files     3808K free
```

In the first case, an old object file may still be there, and TANREN would link it without complaint. After an error, do not link until TATARA has assembled the source without one.

A listing is written on the second pass (chapter 18), so a mistake found on the first pass stops TATARA before any line of the listing is written. `LERR.LST`, named on the command line in section 20.1, is not created, and with `/L` and `/P` instead, `LERR.AS` prints only the message.

20.3 The trail

When the line with the mistake did not come straight from the source named on the command line, the message is followed by a trail: one line for each step that led TATARA to it, starting from the mistake and working outwards. Each line of the trail is indented by four spaces, and has one of these forms:

- `included from FILE(line)`: the file above was included by the `INCLUDE` on that line (chapter 13).
- `in NAME, called from FILE(line)`: the line above is part of the macro `NAME`, and that line used it (chapter 16).
- `in a repeat block, called from FILE(line)`: the line above is part of the `REPT`, `IRP` or `IRPC` block on that line (chapter 17).
- `in a redefined macro, called from FILE(line)`: as for a macro, but the macro has been given a new definition since it was used, so its name no longer belongs to it (section 16.7).

20.3.1 Reading a trail

This example has a mistake in a repeat block, inside the macro `inner`, which is used by the macro `outer`, which is used in an include file. `MACS.INC` defines the two macros:

```
; MACS.INC - two macros, one using the other, with a repeat block.
outer           macro
                nop
                inner
                endm
inner           macro
                rept    1
                ld      (bc),b
                endm
                endm
```

`USE.INC` uses `outer` on its line 2, and `TRAIL.AS` includes both files:

```
; TRAIL.AS - an error in a repeat block, in a macro, in a macro, in an
; include file.
                include macs.inc
                cseg
                include use.inc
                end
```

MSX-DOS

```
MACS.INC(8): ERROR: not a form this instruction has.
    in a repeat block, called from MACS.INC(7)
    in inner, called from MACS.INC(4)
    in outer, called from USE.INC(2)
    included from TRAIL.AS(5)
```

In figure 20.1, each line of the message has a number, and the source line it names has the same number.

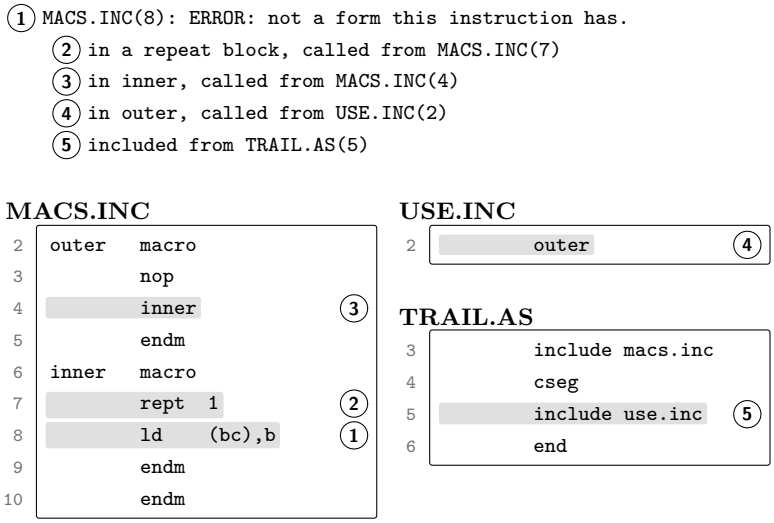


Figure 20.1: Each line of the message and its trail, and the source line it names.

Read the trail from the top down:

1. The mistake is on line 8 of `MACS.INC`, `ld (bc),b`.
2. That line is part of the repeat block that starts on line 7.
3. The repeat block is part of `inner`, used on line 4, inside `outer`.
4. `outer` was used on line 2 of `USE.INC`.

5. `USE.INC` was included by line 5 of `TRAIL.AS`, the source named on the command line.

The first line says where the mistake is, and the last line is always in the source named on the command line. The lines in between show the way from one to the other.

Two things are not in the trail. `USE.INC` gets no `included from` line of its own for line 2, because `called from USE.INC(2)` has already named it. And `include macs.inc` on line 3 does not appear at all: the trail shows how TATARA reached the line with the mistake, and `MACS.INC` had been read to its end long before. Only the definition of the macros came from there.

20.3.2 A redefined macro

A macro that defines itself again while it runs (section 16.7) no longer has a name of its own in the trail:

```
; REDEF.AS - a macro that defines itself again, then fails.
once            macro
once            macro
                endm
                ld      (bc),b
                endm
                cseg
                once
                end
```

MSX-DOS

```
REDEF.AS(5): ERROR: not a form this instruction has.
    in a redefined macro, called from REDEF.AS(8)
```

The name `once` now belongs to the empty macro defined on line 3, so the trail cannot use it for the one that is running.

20.4 Errors found on the second pass

TATARA reads the source twice (section 15.2). A few mistakes can be seen only on the second reading, when TATARA can compare it with the first:

- `phase error - this label had a different value on pass 1.` Something before the label assembled to a different size on the two passes, usually a conditional that gave a different answer (section 15.2).
- `this label was defined on pass 1 and not on pass 2 - a conditional skipped it.` As above, but the label was skipped altogether; the most common cause is a file included twice (section 13.6).
- `a PUBLIC name was never defined.` Found when the object file is finished, at the end of the second pass (chapter 12).

These are the errors after which TATARA deletes the object file it was writing (section 20.2).

20.5 Internal errors

A message that starts with `internal -`, such as `internal - this line emitted more bytes than fit.`, is an *internal error*: TATARA has found a fault in itself, not in the source. Please report it, with:

- the source, and every file it includes;
- the command line;
- everything TATARA printed;
- the output of the same command with `/F`, or with `/M` if the line comes from a macro (section 19.4).

`out of mapper memory.` is not an internal error: the source needs more memory for TATARA's tables than the memory mapper has.

PART III

The linker, Tanren

21 Running TANREN 157

22 Link files 165

23 Where TANREN looks for object files 169

24 How a program is laid out 175

25 Symbols across modules 183

26 Output formats 193

27 Linker error messages 203

CHAPTER 21

Running TANREN

21.1 The command line 157
21.2 The output file 160
21.3 The options 160
21.4 What TANREN prints 161
21.5 The 127-character command line 163
21.6 When TANREN does not start 163

TANREN joins object files made by TATARA into one program. This chapter describes how to run it: the command line, how the output file is named, the options, what TANREN prints, and the messages it gives before it starts. The chapters after it describe the rest in detail: link files (chapter 22), where TANREN looks for object files (chapter 23), how a program is laid out in memory (chapter 24), how names are matched between modules (chapter 25), the kinds of output file (chapter 26), and the error messages (chapter 27).

Like TATARA, TANREN needs MSX-DOS2 or Nextor, and a memory mapper.

21.1 The command line

The command line is **`tanren`**, followed by the names of the object files and any options:

```
tanren [options] object [object...]
```

An object file named without an extension gets `.tro`, so **`tanren main sub`** links `MAIN.TRO` and `SUB.TRO`. The case of the letters does not matter, in names or in options.

These two modules make a small program. `MAIN.AS` calls `greet`, which is in `SUB.AS`. Each includes `MSXDOS.INC`, for the names of the MSX-DOS functions and the `system` macro, and `SUB.AS` also includes `ASCII.INC`, for `CHR_CR` and `CHR_LF`, the carriage return and the line feed that end the line it prints (chapter 14):

```
; MAIN.AS - calls greet, in another module, and ends the program.
                include msxdos.inc
                extrn   greet
                cseg
start:          call    greet
                system  _TERM0
                end     start
```

```
; SUB.AS - greet prints a line.
                include msxdos.inc
                include ascii.inc
                public  greet
                cseg
greet:          ld      de,msg
                system  _STROUT
                ret
                dseg
msg:            db      'Hello from TANREN.',CHR_CR,CHR_LF,'$'
                end
```

Each is assembled into an object file, and TANREN links the two into `MAIN.COM`:

MSX-DOS

```
A:\>tatara /q main.as main.tro
A:\>tatara /q sub.as sub.tro
A:\>tanren main sub
Tatara MSX Linker v1.2.0
Copyright (C) 2026 Javier Lavandeira
https://tatara.tools

2 modules, 16 records, ends at EOF.
Wrote MAIN.COM, 0100-0125 (38 bytes), entry 0100.
A:\>main
Hello from TANREN.
```

21.1.1 The order of the object files

TANREN places the modules in memory in the order they are named: the code of the first at memory address 0100h, the code of the next straight after it, and so on (chapter 24 describes where the data goes). MSX-DOS always starts a `.COM` program at memory address 0100h, so the module whose code must run first has to be named first.

Named the other way round, the same two modules make a program that starts with the code of `greet`:

MSX-DOS

```
A:\>tanren sub main
...
WARNING: SUB.COM is entered at 0100, not at 0109.
2 modules, 16 records, ends at EOF.
Wrote SUB.COM, 0100-0125 (38 bytes), entry 0109.
```

The summary line says that `start`, the address on the `END` line of `MAIN.AS`, is now at 0109h, but a `.COM` program cannot start there: MSX-DOS runs whatever is at 0100h. TANREN warns about it before the summary, even with `/Q`, and still writes the file. The warning comes whenever a file written without `/B` has an entry address other than its first address. With `/B`, the entry address goes into the `BLOAD` header and there is no warning (chapter 26).

21.1.2 Spaces between names

The names and the options are separated by spaces or tabs. Anything else, a comma included, is part of the name:

MSX-DOS

```
A:\>tanren main,sub
...
ERROR: cannot open MAIN,SUB.TRO
```

An option can go anywhere on the line, before the names, after them or between them.

21.2 The output file

TANREN names the output file after the first object file, with the extension `.com`, in the same directory as that object file. With `/B`, which writes a file for `BLOAD`, the extension is `.bin` instead:

MSX-DOS

```
A:\>tanren /b main sub
...
Wrote MAIN.BIN, 0100-0125 (38 bytes), BLOAD header, entry 0100.
```

`/O:` names the output file, with the name written straight after the colon:

MSX-DOS

```
A:\>tanren main sub /o:prog.com
...
Wrote PROG.COM, 0100-0125 (38 bytes), entry 0100.
```

The name is used exactly as written. TANREN adds no extension to it, so **`/o:prog`** writes a file called `PROG`, with no extension, which MSX-DOS will not run as a command.

The output file may not be one of the object files, so that a mistake on the command line cannot overwrite an input:

MSX-DOS

```
A:\>tanren /o:sub.tro main sub
...
ERROR: the output file SUB.TRO is also an input file.
```

TANREN compares the names as they are written, so it does not notice the same file reached by two different paths.

21.3 The options

Table 21.1 lists TANREN's options. An address is one to four hexadecimal digits, with no `h`, as in **`/p:4000`**. `/D:`, `/O:` and `/P:` need the colon: **`/p`** alone is an unknown option.

`/P:` changes where the code starts, and the summary line shows it:

MSX-DOS

Option	What it does	Chapter
`/B`	writes a `BLOAD` header on the output file, whose default extension becomes `.bin`	26
`/D:addr`	starts the data segments at `addr`	24
`/M`	prints the segment and group tables, and the symbols	25
`/O:file`	names the output file	this chapter
`/P:addr`	starts the code segments at `addr`	24
`/Q`	leaves out the banner and the summary line	this chapter
`/R`	prints every record in the object files, for diagnosis	appendix F
`/V`	prints the banner, and stops	this chapter
`/?`	prints the usage screen, and stops	this chapter

Table 21.1: TANREN's options.

```
A:\>tanren /p:4000 /o:high.bin main sub
...
Wrote HIGH.BIN, 4000-4025 (38 bytes), entry 4000.
```

21.4 What TANREN prints

TANREN starts with a banner, as TATARA does, and ends with a summary of two lines:

MSX-DOS

```
2 modules, 16 records, ends at EOF.
Wrote MAIN.COM, 0100-0125 (38 bytes), entry 0100.
```

- **The modules** are the object files TANREN read, and **the records** are the pieces of information in them: the bytes of each segment, the names, and the places where TANREN has to fill in an address (appendix F).
- **ends at EOF** says that each object file ended where its last record said it

would. If one has anything after that, the line says `AND BYTES AFTER THEM.` instead, and a line before it names the file. TATARA does not make such a file, so assemble that module again (section 27.3).

- **Wrote** names the output file, the addresses of its first and last bytes, and how many bytes that is. With `/B`, the file on the disk is seven bytes longer than that, because of the `BLOAD` header, and the line says so.
- **entry** is the address named on an `END` line. It is written into the header by `/B`. Without `/B`, it should be the first address of the file, or TANREN warns.

When none of the modules has any bytes to write, the second line is `Nothing to write: no module has any content.` instead.

`/Q` leaves out the banner and the summary. Errors are printed with or without it, so a link with `/Q` that prints nothing has succeeded. `/V` prints the banner and stops, and `/?`, or **`tanren`** with no object file at all, prints the banner and the usage screen:

MSX-DOS

```
A:\>tanren
Tatara MSX Linker v1.2.0
Copyright (C) 2026 Javier Lavandeira
https://tatara.tools

Usage:  TANREN [options] <object|@list> [more...]

Options:  /B Write a BLOAD header on the output file
          /D:<addr> Start the data segments at <addr>
          /M Print the segment and group tables
          /O:<file> Write the image to <file>
          /P:<addr> Start the code segments at <addr>
          /Q Omit the banner above and the summary line
          /R Print every record in the object files
          /V Print the program version
          /? Print this screen

<object>  : A .tro file; .tro is assumed when you leave it off.
@<list>   : A file of the same words; ; is a comment, .lnk
            is assumed.
<addr>    : One to four hex digits, as in /P:4000.
<file>    : The output. Default: the first object with .com
            (.bin with /B). It may not be an input file.
```

21.5 The 127-character command line

MSX-DOS gives a command 127 characters, counting the name of the program and the spaces. At the prompt, COMMAND2 accepts no more: every key after the 127th only beeps. In a batch file, a longer line is not run at all, and COMMAND2 prints `*** Command too long`.

A program of many modules can need more than that. A *link file* holds the same words in a file, as many as needed, and the command line names only the file: **`tanren @prog`** (chapter 22).

If some other program starts TANREN with a longer line, MSX-DOS cuts it at 127 characters without saying so. When TANREN's command line is exactly 127 characters long, it cannot tell whether anything was cut, and it warns, even with `/Q`:

MSX-DOS

```
WARNING: the command line is 127 characters, which is all
MSX-DOS gives - anything past it was dropped in silence.
Use @<file> if a module is missing.
```

21.6 When TANREN does not start

Table 21.2 lists the messages TANREN prints before it links anything. The first three come before the banner.

A mistake found while linking, such as a name that no module defines, is described in chapter 27.

Message	Meaning
`ERROR: Tanren needs MSX-DOS2 or Nextor.`	The MSX is running MSX-DOS1.
`ERROR: Tanren needs a memory mapper.`	The MSX has no memory mapper.
`ERROR: unknown option.`	An option TANREN does not have, or `/D`, `/O` or `/P` without a colon.
`ERROR: cannot open NAME.TRO`	No object file of that name was found (chapter 23 describes where TANREN looks).
`ERROR: the output file NAME is also an input file.`	The output file would overwrite an object file.

Table 21.2: Messages before linking.

CHAPTER 22

Link files

22.1 A first link file 165
22.2 Writing a link file 166
22.3 Link files and the command line 167
22.4 Messages 168

A *link file* holds the words of a TANREN command line, the names of the object files and the options, in a text file. **`tanren @prog`** reads the file `PROG.LNK` and takes its words as if they had been typed where **`@prog`** stands. A link file has no limit of 127 characters (section 21.5), and it keeps the list of a program's modules in one place, with comments, so that every link uses the same list.

22.1 A first link file

This link file links the program of chapter 21:

```
; PROG.LNK - the two modules of the example program.

main            ; the start of the program: it must come first
sub             ; greet

/o:prog.com     ; the output file
```

It names the two object files, `MAIN.TRO` and `SUB.TRO`, and the output file, `PROG.COM`. The command line names only the link file:

MSX-DOS

```
A:\>tanren @prog
Tatara MSX Linker v1.2.0
Copyright (C) 2026 Javier Lavandeira
https://tatara.tools

2 modules, 16 records, ends at EOF.
Wrote PROG.COM, 0100-0125 (38 bytes), entry 0100.
A:\>prog
Hello from TANREN.
```

22.2 Writing a link file

A link file is a text file, written with any editor. It holds the same words as a command line, with these rules:

- Words are separated by spaces, tabs or the ends of lines. A line can hold one word, several, or none: `ONELINE.LNK`, which holds `main sub /o:one.com` on a single line, does the same as `PROG.LNK`, except for the name of the output file.
- A semicolon starts a comment, which runs to the end of the line. A comment can have a line of its own or follow the words of a line.
- An object file named without an extension gets `.tro`, as on the command line.
- Options are words like any other, and can be anywhere in the file.
- The file ends at its last character, or at a Ctrl+Z if it has one. The last line does not need to end with a line end.

The name after `@` gets `.lnk` if it has no extension, so **`tanren @prog`** and **`tanren @prog.lnk`** read the same file. TANREN opens the link file with the name as written, in the current directory unless the name includes a path.

22.2.1 A program too big for the command line

TATARA itself is made of nineteen modules, and is linked with this link file (its first lines, a comment, are left out here):

```
tatara.tro                              ; the driver
cmdline.tro srcline.tro fields.tro      ; the front
dirtab.tro macros.tro expand.tro        ; directives and macros
cond.tro expr.tro                       ; conditionals and expressions
symtab.tro optab.tro insn.tro           ; the tables and the back end
emit.tro objout.tro errs.tro

msxdos.tro strutil.tro alloc.tro hash.tro       ; shared
```

The names of the nineteen object files alone take 198 characters, with a space between each, far more than the 127 of a command line. With the link file, the command that makes `TATARA.COM` is short:

```
tanren /o:build\tatara.com @tatara.lnk
```

The comments also say what each group of modules is, which a command line cannot.

22.3 Link files and the command line

The words of a link file take the place of the `@` word, so a command line can mix them with names and options of its own. TANREN then treats every word the same, whether it was typed or read from the file.

- **The order of the modules** is the order of the words, as always (section 21.1). `SUBONLY.LNK` names only `sub`. **`tanren main @subonly`** puts `MAIN.TRO` first, and **`tanren @subonly main`** puts `SUB.TRO` first:

  MSX-DOS

  ```
  A:\>tanren main @subonly
  ...
  Wrote MAIN.COM, 0100-0125 (38 bytes), entry 0100.
  A:\>tanren @subonly main /o:rev.com
  ...
  WARNING: REV.COM is entered at 0100, not at 0109.
  2 modules, 16 records, ends at EOF.
  Wrote REV.COM, 0100-0125 (38 bytes), entry 0109.
  ```

  `REV.COM` starts with the code of `greet`, and TANREN warns about it (section 21.1).

- **Options** can come before or after the `@` word. **`tanren /p:4000 @prog /o:high.bin`** starts the code at 4000h and writes `HIGH.BIN`.

- **An option given twice**: the last one counts. `PROG.LNK` has `/o:prog.com`, so a `/O:` after **`@prog`** replaces it, and one before is replaced by it:

MSX-DOS

```
A:\>tanren @prog /o:other.com
...
Wrote OTHER.COM, 0100-0125 (38 bytes), entry 0100.
A:\>tanren /o:first.com @prog
...
Wrote PROG.COM, 0100-0125 (38 bytes), entry 0100.
```

- **Only one link file** can be named on a command line, and a link file cannot name another.

TANREN looks for the object files that a link file names in the link file's own directory first. Chapter 23 describes where it looks.

22.4 Messages

Table 22.1 lists the messages about link files. They come before the banner, because TANREN reads the link files before it starts. The messages call a link file a "file list".

Message	Meaning
`ERROR: cannot open the file list NAME.LNK`	There is no link file of that name, or `@` has no name after it.
`ERROR: only one file list may be given.`	The command line names more than one link file.
`ERROR: a file list may not name another one.`	A word in a link file starts with `@`.
`ERROR: too many words on the command line or in the file list.`	The words take more than 4096 characters in all, or one word is longer than 128 characters.

Table 22.1: Messages about link files.

A module named twice, for example on the command line and again in the link file, is found while linking: its public names are defined twice (chapter 27).

CHAPTER 23

Where TANREN looks for object files

23.1 Three places 169
23.2 The TANREN environment variable. 171
23.3 Paths in the name 172
23.4 Where the output file goes. 172
23.5 Messages 172

An object file named without a drive or a directory, such as `SUB.TRO`, can be in more than one place. This chapter describes where TANREN looks for it, how the `TANREN` environment variable fits in, and where the output file goes when an object file is found somewhere other than the current directory.

TANREN looks for object files the way TATARA looks for include files (chapter 13), with the link file in the place of the file that holds the `INCLUDE` line.

23.1 Three places

TANREN looks for an object file in three places, in this order, and uses the first one it finds:

1. the directory of the link file that names it, if it was named in a link file (chapter 22);
2. the current directory;
3. each directory named in the `TANREN` environment variable, from left to right.

Figure 23.1 shows the three places for one command. `PROG.LNK`, in `BUILD`, names `main` and `sub`. TANREN takes `SUB.TRO` from `BUILD`, beside the link file, and never reaches the copies in the current directory or in `TATARA\LIB`. There is no `MAIN.TRO` in `BUILD`, so it takes the one in the current directory.

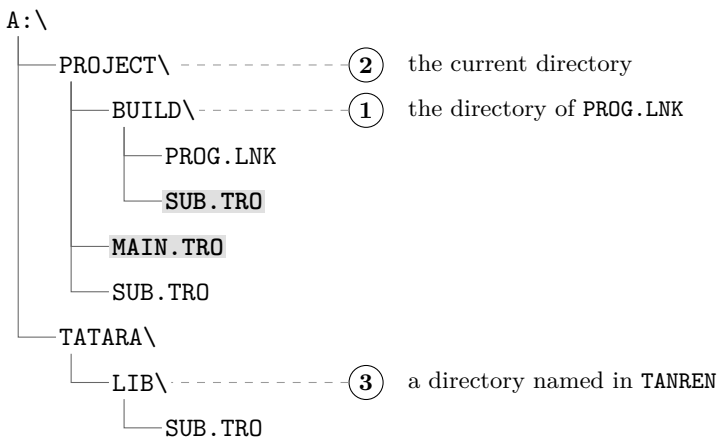


Figure 23.1: The three places TANREN looks for an object file, in order. The files TANREN uses are shaded.

The directory of the link file comes first so that a program's own object files, kept beside its link file, always win over other files of the same name. An object file named on the command line, and not in a link file, starts with the second place.

The examples in this chapter use several copies of `SUB.TRO`, each made from a source that prints the directory it was in, so that running the program shows which copy TANREN used. With the link file `PROG.LNK` in `LNK`, and a `SUB.TRO` beside it:

MSX-DOS

```
A:\>tanren @lnk\prog /o:lnk.com
...
2 modules, 16 records, ends at EOF.
Wrote LNK.COM, 0100-0123 (36 bytes), entry 0100.
A:\>lnk
SUB.TRO from LNK
```

23.2 The TANREN environment variable

`TANREN` names directories where TANREN looks for object files that are not in the first two places, such as a directory of modules that several programs share. It works like `TATARA` (section 13.3): several directories are separated by semicolons and tried from left to right, and a `\` at the end of each is not needed, but does no harm. In `AUTOEXEC.BAT` or another batch file:

```
set tanren=a:\tatara\lib;a:\project\mod
```

At the prompt, the value goes between quotation marks, as in chapter 3.

Without `TANREN`, an object file that is not in the first two places is not found:

MSX-DOS

```
A:\>tanren main sub
...
ERROR: cannot open SUB.TRO
```

With `TANREN` set to the directory that holds it, the same command links:

MSX-DOS

```
A:\>tanren main sub
...
2 modules, 16 records, ends at EOF.
Wrote MAIN.COM, 0100-0123 (36 bytes), entry 0100.
A:\>main
SUB.TRO from LIB
```

A directory in `TANREN` that does not exist is passed over, and the next one is tried.

`TANREN` is the last of the three places. A `SUB.TRO` in the current directory is used before the one in a `TANREN` directory:

MSX-DOS

```
A:\>tanren main sub /o:cur.com
...
Wrote CUR.COM, 0100-0135 (54 bytes), entry 0100.
A:\>cur
SUB.TRO from the current directory
```

23.3 Paths in the name

A name that starts with a drive, such as `B:SUB`, or with `\`, such as `\PROJECT\SUB`, is an *absolute path*. TANREN opens it as written and looks nowhere else, even if `TANREN` names a directory that holds the file:

MSX-DOS

```
A:\>tanren main b:sub /o:drv.com
...
ERROR: cannot open B:SUB.TRO
```

Any other name is a *relative path*, and is looked for in each of the three places in turn. **`tanren main lib\sub`** finds `LIB\SUB.TRO` under the current directory, with no `TANREN` at all.

23.4 Where the output file goes

Without `/O:`, the output file is named after the first object file, with `.com` (chapter 21), and it goes into the directory where that object file was found. When the first object file comes from a `TANREN` directory or from the link file's directory, so does the output file:

MSX-DOS

```
A:\>tanren sub main
...
WARNING: LIB\SUB.COM is entered at 0100, not at 0109.
2 modules, 16 records, ends at EOF.
Wrote LIB\SUB.COM, 0100-0123 (36 bytes), entry 0109.
```

Here `SUB.TRO` was found through `TANREN`, in `LIB`, and `SUB.COM` was written there too, not in the current directory. The summary line shows the path. (The warning is about the order of the modules, as in section 21.1: `SUB.TRO` is named first.) To be sure where the output file goes, name it with `/O:`.

23.5 Messages

Table 23.1 lists the messages about finding object files.

Message	Meaning
`ERROR: cannot open NAME.TRO`	The object file is in none of the three places, or not where an absolute path says. The name is shown as it was written, in capitals.
`ERROR: the TANREN variable is too long.`	`TANREN` is longer than 127 characters.

Table 23.1: Messages about finding object files.

CHAPTER 24

How a program is laid out

24.1 Code, then data 175
24.2 Named segments 177
24.3 Moving the code and the data: /P: and /D:. 178
24.4 Transient segments across modules 179
24.5 Absolute segments 180
24.6 Messages 181

TATARA gives every label an address inside its segment, starting from 0 (chapter 10). TANREN decides where each segment of each module goes in memory, and so gives every label its final address. This chapter describes the rules it follows, the options that change them, and what ends up in the output file.

24.1 Code, then data

TANREN places all the code first and all the data after it:

1. The code segments start at memory address 0100h, where MSX-DOS loads a `.COM` program.
2. The data segments start straight after the last byte of code.

A segment used by several modules becomes one piece, with each module's part in the order the modules are named on the command line.

These two modules each have some code and some data. `A.AS` also has a named code segment, `extra`:

```
; A.AS - code, data, and a named code segment.
                public  astart,aextra,adata
                cseg
astart:         ld      hl,adata
                ret
                cseg    extra
aextra:         nop
                dseg
adata:          ds      4
                end     astart
```

```
; B.AS - code and data.
                public  bcode,bdata
                cseg
bcode:          ld      hl,bdata
                ret
                dseg
bdata:          db      1,2
                end
```

`/M` makes TANREN print, among other things, the segments of the program with the address each one was given:

MSX-DOS

```
A:\>tanren /m a b
...
Modules: 2
Groups:
Segments:
  flags 00 group FF base 0100 size 0008 end 0107  C
  flags 00 group FF base 0108 size 0001 end 0108 extra
  flags 01 group FF base 0109 size 0006 end 010E  D
Symbols:
  D- 0100 astart
  D- 0108 aextra
  D- 0104 bcode
  D- 0109 adata
  D- 010D bdata
2 modules, 21 records, ends at EOF.
Wrote A.COM, 0100-010E (15 bytes), entry 0100.
```

Each line under `Segments:` is one segment, in the order TANREN placed them. The last word is the segment's name: `C` is the default code segment, `D` the default data segment, and any other name is a named segment. `base` is its first address, `size` its size in bytes, and `end` its last address. `flags` is `00` for code and `01` for data. The groups and the symbols, and the rest of `/M`, are described in chapter 25.

Figure 24.1 shows the same program in memory. The code segment holds `A`'s four bytes and then `B`'s, from 0100h to 0107h; `extra` follows at 0108h; and the data

segment starts at 0109h, with A's four bytes and then B's two.

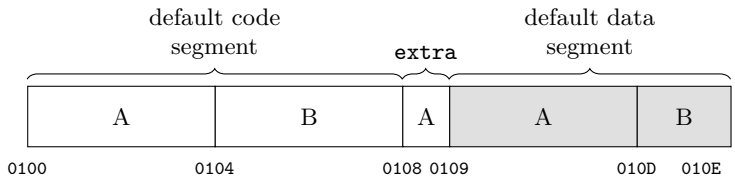


Figure 24.1: The memory of `tanren a b`. Each box is one module's part of a segment; the shaded ones are data.

Naming the modules the other way round, **`tanren b a`**, puts B's part first in each segment: `bcode` moves to 0100h, `astart` to 0104h, and the entry address with it.

24.1.1 What goes into the file

The output file holds the memory from the first byte that has content to the last one. A `DS` reserves memory without giving it any content (chapter 11), so a `DS` at the end of the program is not written to the file at all. With **`tanren b a`**, A's `ds 4` comes last, and the file ends at 010Ah, after B's two bytes:

MSX-DOS

```
A:\>tanren /m b a /o:ba.com
...
WARNING: BA.COM is entered at 0100, not at 0104.
2 modules, 21 records, ends at EOF.
Wrote BA.COM, 0100-010A (11 bytes), entry 0104.
```

The same applies to the first bytes. The memory a `DS` leaves out of the file is not cleared either: when the program starts, it holds whatever was there before. A program that needs a variable to start at 0 must set it to 0 itself, or define it with `DB` or `DW`.

24.2 Named segments

A program can have several segments of each kind (section 10.5). Within each kind, TANREN places the default segment first, and then the named segments in the order they first appear in the modules, taken in command-line order. This module selects its named segments before the default ones:

```
; ORDER.AS - named segments written before the default ones.
                public  zfirst,cmain,zdata,dmain
                cseg    zcode
zfirst:         nop
                cseg
cmain:          ret
                dseg    zvars
zdata:          db      1
                dseg
dmain:          db      2
                end     cmain
```

MSX-DOS

```
A:\>tanren /m order
...
Segments:
  flags 00 group FF base 0100 size 0001 end 0100  C
  flags 00 group FF base 0101 size 0001 end 0101 zcode
  flags 01 group FF base 0102 size 0001 end 0102  D
  flags 01 group FF base 0103 size 0001 end 0103 zvars
```

The default code segment still starts at 0100h. This matters for a `.COM` program, which MSX-DOS always starts at 0100h: the code that runs first belongs in the default code segment of the first module.

24.3 Moving the code and the data: /P: and /D:

`/P:` starts the code at another address, and the data follows it as before:

MSX-DOS

```
A:\>tanren /m /p:8000 a b /o:p8000.bin
...
Segments:
  flags 00 group FF base 8000 size 0008 end 8007  C
  flags 00 group FF base 8008 size 0001 end 8008 extra
  flags 01 group FF base 8009 size 0006 end 800E  D
...
Wrote P8000.BIN, 8000-800E (15 bytes), entry 8000.
```

MSX-DOS loads a `.COM` program at 0100h, so a program linked for another address is not a `.COM` program: it is loaded in some other way, for example by MSX-BASIC's `BLOAD` (chapter 26).

`/D:` starts the data at an address of its own, instead of straight after the code. The data can be above the code or below it. With **`/p:c000 /d:8000`**, the code is at

C000h and the data under it at 8000h:

MSX-DOS

```
A:\>tanren /m /p:c000 /d:8000 a b /o:high.bin
...
Segments:
  flags 00 group FF base C000 size 0008 end C007  C
  flags 00 group FF base C008 size 0001 end C008 extra
  flags 01 group FF base 8000 size 0006 end 8005  D
...
Wrote HIGH.BIN, 8004-C008 (16389 bytes), entry C000.
```

The file starts at 8004h, not 8000h, because the first four bytes of data are `A`'s `ds 4`, which have no content (section 24.1.1).

The data cannot start inside the code: **`/d:0102`**, with code from 0100h to 0108h, stops TANREN with `/D: would put the data on top of the code.`

Warning. Everything between the code and the data is in the file, written as zeros. With **`/d:c000`**, a `.COM` program of fifteen bytes of code and data becomes a file of 48,902 bytes:

MSX-DOS

```
A:\>tanren /m /d:c000 a b /o:dc000.com
...
Wrote DC000.COM, 0100-C005 (48902 bytes), entry 0100.
```

A data segment made only of `DS` has no content, and adds nothing to the file wherever it is placed.

24.4 Transient segments across modules

The groups of a transient data segment (section 10.6) share the same memory, and TANREN applies this across modules: every group of a transient segment, in whichever module, starts at the same address, and the segment takes the size of its largest group. These two modules each add a group to the segment `scratch`:

```
; T1.AS - the group reading of the transient segment scratch.
                public  inbuf
                dseg    scratch,transient
                group   reading
inbuf:          ds      64
                end
```

```
; T2.AS - the group writing of the same segment.
                public  outbuf
                dseg    scratch,transient
                group   writing
outbuf:         ds      32
                end
```

MSX-DOS

```
A:\>tanren /m a b t1 t2 /o:tr.com
...
Groups:
  reading
  writing
Segments:
  flags 00 group FF base 0100 size 0008 end 0107  C
  flags 00 group FF base 0108 size 0001 end 0108 extra
  flags 01 group FF base 0109 size 0006 end 010E  D
  flags 03 group 00 base 010F size 0040 end 014E scratch
  flags 03 group 01 base 010F size 0020 end 012E scratch
...
Wrote TR.COM, 0100-010E (15 bytes), entry 0100.
```

A transient segment has one line for each group, with `flags 03`, and the number of the group in `Groups:`, counting from 00. Both groups start at 010Fh (figure 24.2), so `inbuf` and `outbuf` have the same address, and `scratch` takes 64 bytes, the size of `reading`. It holds only `DS`, so the file is still fifteen bytes long.

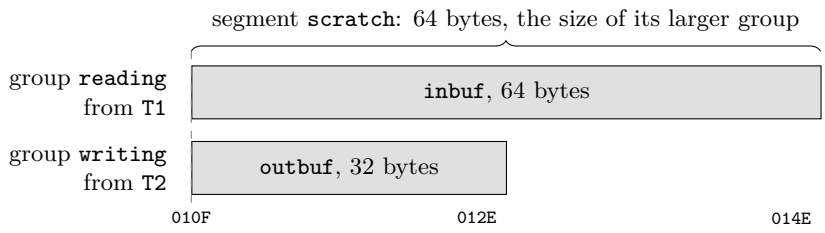


Figure 24.2: Two groups of the transient segment `scratch`, from two modules, at the same address.

24.5 Absolute segments

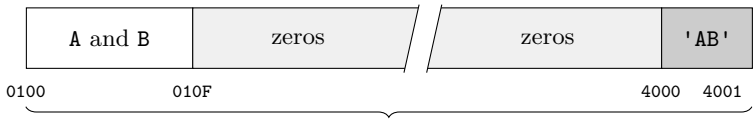
An absolute segment (chapter 10) is not moved: its bytes go at the addresses written in the source. This module puts two bytes at 4000h:

```
; ABS.AS - two bytes at 4000h.
                public  romhdr
                aseg
                org     4000h
romhdr:         db      'AB'
                end
```

Linked with `A` and `B`, it makes the program run from 0100h to 4001h, and everything between the last byte of data and 4000h is written as zeros (figure 24.3):

MSX-DOS

```
A:\>tanren /m a b abs /o:gap.com
...
Wrote GAP.COM, 0100-4001 (16130 bytes), entry 0100.
```



`GAP.COM`: 16,130 bytes, from 0100h to 4001h

Figure 24.3: An absolute segment far above the code: the gap is in the file, as zeros.

Absolute segments are meant for programs that have to put bytes at fixed addresses, such as a ROM image (chapter 26). In a `.COM` program, bytes far above the code make a large file.

24.6 Messages

Table 24.1 lists the messages about the layout.

Message	Meaning
`ERROR: /D: would put the data on top of the code.`	The address given with `/D:` is inside the code.
`ERROR: the linked image would run past FFFFh.`	The program does not fit below FFFFh from the address it starts at, for example 300 bytes of code with **`/p:ff00`**.

Table 24.1: Messages about the layout.

CHAPTER 25

Symbols across modules

25.1 How names are matched 183
25.2 What an external's value can be 185
25.3 Names that no module defines 186
25.4 Names defined twice. 187
25.5 Capitals and small letters 188
25.6 Every module is linked 189
25.7 /M in full 190
25.8 Messages 191

A module offers names to the others with `PUBLIC`, and uses theirs with `EXTRN` (section 12.5). TATARA leaves every external name unfinished, and TANREN finishes it with the value the other module gave it. This chapter describes how TANREN matches the names, what it does with names that no module defines or that two modules define, and what `/M` prints.

25.1 How names are matched

In the program of chapter 21, `MAIN.AS` declares `greet` with `EXTRN` and calls it, and `SUB.AS` declares it with `PUBLIC` and defines it:

```
; MAIN.AS - calls greet, in another module, and ends the program.
                include msxdos.inc
                extrn   greet
                cseg
start:          call    greet
                system  _TERM0
                end     start
```

```
; SUB.AS - greet prints a line.
                include msxdos.inc
                include ascii.inc
                public  greet
                cseg
greet:          ld      de,msg
                system  _STROUT
                ret
                dseg
msg:            db      'Hello from TANREN.',CHR_CR,CHR_LF,'$'
                end
```

TANREN keeps one table of names for the whole link, and reads the modules twice:

1. **On the first pass**, it reads every module's public and external names into the table. A public name is *defined*, with its value: `greet` is the first byte of `SUB`'s code, and `SUB`'s code comes after `MAIN`'s eight bytes, so `greet` is at 0108h. An external name is *referenced*: some module wants its value. At the end of the pass, every referenced name must be defined.
2. **On the second pass**, it writes the bytes of each module, and fills in every place where an external name is used with the value from the table.

Figure 25.1 follows `greet` through the two passes. TATARA wrote `call greet` as `CD 00 00`, and TANREN turns it into `CD 08 01`, a call to 0108h.

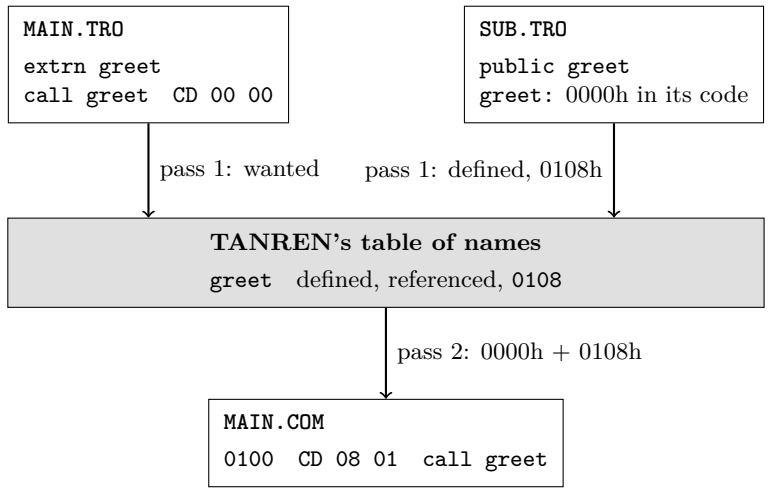


Figure 25.1: How `greet`, external in `MAIN` and public in `SUB`, gets its value.

All the names are in the table before any byte is written, so the order of the modules does not matter for the names: a module can use a name defined in a

module named before it or after it. The order matters only for where each module goes in memory (chapter 24).

`/M` prints the table, among other things. For this program it has one name:

MSX-DOS

```
A:\>tanren /m main sub
...
Symbols:
  DR 0108 greet
2 modules, 16 records, ends at EOF.
Wrote MAIN.COM, 0100-0125 (38 bytes), entry 0100.
```

`D` says that `greet` is defined, `R` that it is referenced, and 0108 is its value. Section 25.7 describes the rest of `/M`.

25.2 What an external's value can be

An external name can have a number added to it or taken away from it (section 8.6). TATARA writes that number in the place where the value goes, and TANREN adds the name's value to it. These two modules use names in the code and in the data:

```
; USE.AS - uses two names from VALS.AS, with and without an addend.
                extrn   table,count
                cseg
start:          ld      hl,table+2
                ld      bc,count
                ld      de,count+1
                ret
                dseg
ptrs:           dw      table,table+4
                end     start
```

```
; VALS.AS - a table in the data segment, and an absolute value.
                public  table,count
count           equ     10
                dseg
table:          db      1,2,3,4,5,6
                end
```

MSX-DOS

```
A:\>tanren /m use vals
...
Symbols:
  DR 000A count
  DR 010E table
2 modules, 17 records, ends at EOF.
Wrote USE.COM, 0100-0113 (20 bytes), entry 0100.
```

`table` is at 010Eh, after the data of `USE`, so in `USE.COM` `ld hl,table+2` is `21 10 01`, and `dw table,table+4` is `0E 01 12 01`.

`count` is defined with `EQU`, so it is not an address in a segment but an *absolute* value (section 8.6). TANREN does not move it: it is 000Ah wherever the modules go, and `ld de,count+1` is `11 0B 00`.

25.3 Names that no module defines

A name that is referenced and never defined stops TANREN at the end of the first pass, and no output file is written. Linked alone, `MAIN` has nobody to define `greet`:

MSX-DOS

```
A:\>tanren main
...
Never defined:
  greet
ERROR: the symbols above were never defined.
```

TANREN lists every name that is missing, not only the first, so that one run shows them all. This module uses three names, and only `beta` is defined, in `BETA.AS`:

```
; MISS.AS - three externals; nobody defines alpha or gamma.
                extrn   alpha,beta,gamma
                cseg
                ld      hl,alpha
                ld      de,beta
                ld      bc,gamma
                ret
                end
```

```
; BETA.AS - defines beta.
                public  beta
                cseg
beta:           ret
                end
```

MSX-DOS

```
A:\>tanren miss beta
...
Never defined:
  gamma
  alpha
ERROR: the symbols above were never defined.
```

The names are listed in no particular order.

Warning. A name declared with `EXTRN` is referenced even if the module never uses it. TANREN still wants a definition for it:

```
; UNUSED.AS - declares an external and never uses it.
                extrn   nothere
                cseg
start:          ret
                end     start
```

MSX-DOS

```
A:\>tanren unused
...
Never defined:
  nothere
ERROR: the symbols above were never defined.
```

When a module stops using an external name, remove it from the `EXTRN` line too.

25.4 Names defined twice

A name can be defined by only one module. The second definition stops TANREN while it reads the modules, and the line before the error names the name and the module where the second definition was found. Here `DUP2.AS` defines a second `greet`, and was assembled into `OTHER.TRO`:

```
; DUP2.AS - a second greet.
                public  greet
                cseg
greet:          ret
                end
```

MSX-DOS

```
A:\>tatara /q dup2.as other.tro
A:\>tanren main sub other /o:dup.com
...
  greet is defined again in module dup2
ERROR: that public symbol is already defined.
```

The module is named after the source it was assembled from, not after the object file, so the message says `dup2`, not `other`.

The same happens when one object file is named twice, for example on the command line and in a link file (chapter 22):

MSX-DOS

```
A:\>tanren main sub sub /o:dup.com
...
  greet is defined again in module sub
ERROR: that public symbol is already defined.
```

TANREN stops at the first name defined twice, and does not look for missing names after it. With both mistakes in one link, correct the name defined twice first, and link again.

25.5 Capitals and small letters

Without `/C`, TATARA does not tell capitals from small letters in names (section 7.3), and neither does TANREN. This module calls `GREET`, in capitals:

```
; UPPER.AS - calls GREET, in capitals.
                include msxdos.inc
                extrn   GREET
                cseg
start:          call    GREET
                system  _TERM0
                end     start
```

It links with `SUB`, whose name is `greet`. `/M` shows the name as TANREN first saw it:

MSX-DOS

```
A:\>tanren /m upper sub
...
Symbols:
  DR 0108 GREET
2 modules, 16 records, ends at EOF.
Wrote UPPER.COM, 0100-0125 (38 bytes), entry 0100.
```

Assembled with `/C`, the two modules have two different names, one defined and one only referenced, and the link fails:

MSX-DOS

```
A:\>tatara /q /c upper.as upperc.tro
A:\>tatara /q /c sub.as subc.tro
A:\>tanren /m upperc subc
...
Symbols:
  D- 0108 greet
  -R ---- GREET
Never defined:
  GREET
ERROR: the symbols above were never defined.
```

A module assembled with `/C` and one assembled without it cannot be linked together (section 7.3):

MSX-DOS

```
A:\>tanren upperc sub /o:mix.com
...
ERROR: one module was assembled /C and another was not.
```

25.6 Every module is linked

TANREN links exactly the modules it is given. A module that no other module refers to is still linked, and its bytes are in the output file. Nothing refers to `spare`, in `EXTRA.AS`:

```
; EXTRA.AS - a module nobody refers to.
                public  spare
                cseg
spare:          ld      a,1
                ret
                end
```

MSX-DOS

```
A:\>tanren /m main sub extra /o:ex.com
...
Symbols:
  D- 0111 spare
  DR 0108 greet
3 modules, 22 records, ends at EOF.
Wrote EX.COM, 0100-0128 (41 bytes), entry 0100.
```

`spare` is defined and not referenced, and its three bytes make `EX.COM` three bytes longer than `MAIN.COM`. TANREN does not search a library for the modules a program needs: every module named on the command line or in the link file is linked, whether it is used or not.

25.7 /M in full

`/M` prints four parts, after the banner and before the summary line. For **`tanren /m main sub`**:

MSX-DOS

```
A:\>tanren /m main sub
Tatara MSX Linker v1.2.0
Copyright (C) 2026 Javier Lavandeira
https://tatara.tools

Modules: 2
Groups:
Segments:
  flags 00 group FF base 0100 size 0011 end 0110  C
  flags 01 group FF base 0111 size 0015 end 0125  D
Symbols:
  DR 0108 greet
2 modules, 16 records, ends at EOF.
Wrote MAIN.COM, 0100-0125 (38 bytes), entry 0100.
```

- **Modules** is how many object files TANREN read.
- **Groups** lists the groups of the transient segments, one name to a line, numbered from 00 in the order they are listed (section 24.4). This program has none.
- **Segments** lists every segment, where it was placed and its size (section 24.1).
- **Symbols** lists every public and external name, one to a line, in no particular order. The first column is two letters: `D` if a module defines the name, and

`R` if a module declares it external, with `-` in place of either letter that does not apply. Then comes the value, or `----` for a name no module defines, and then the name.

`/M` prints its tables even when names are missing, before the list of missing names. A missing name is `-R ----`:

MSX-DOS

```
A:\>tanren /m main /o:m1.com
...
Modules: 1
Groups:
Segments:
  flags 00 group FF base 0100 size 0008 end 0107  C
  flags 01 group FF base 0108 size 0000 end -----  D
Symbols:
  -R ---- greet
Never defined:
  greet
ERROR: the symbols above were never defined.
```

A segment of size 0000 has no last address, and `end` shows `-----`.

25.8 Messages

Table 25.1 lists the messages about names. Chapter 27 lists all of TANREN's messages.

Message	Meaning
`ERROR: the symbols above were never defined.`	The names listed under `Never defined:` are declared external by some module and defined by none.
`ERROR: that public symbol is already defined.`	The line before it, `NAME is defined again in module MOD`, names a public name that a second module defines too.
`ERROR: one module was assembled /C and another was not.`	All the modules must be assembled with `/C`, or all without it.

Table 25.1: Messages about names.

CHAPTER 26

Output formats

26.1 What TANREN writes. 193
26.2 A .COM program 194
26.3 A raw image for another address 194
26.4 A file for BLOAD: /B 195
26.5 The entry address 196
26.6 A ROM image. 198
26.7 How large a program can be 199
26.8 When there is nothing to write. 200
26.9 Messages 200

TANREN writes one output file for each link. This chapter describes what is in that file, the kinds of file it can be, and how to link a program for each of them: a `.COM` program for MSX-DOS, an image for some other address, a file for MSX-BASIC's `BLOAD`, and a ROM image. It ends with how large a program can be.

26.1 What TANREN writes

The output file holds the program's memory from the first byte that has content to the last, and nothing else (section 24.1.1). The file does not say at which address it goes, except in the header that `/B` adds: whatever loads the file has to know.

Table 26.1 lists the four kinds of file, and how each one is made. Only `.COM` is TANREN's default; for the others, the output file is named with `/O:`, or gets `.bin` with `/B`.

Kind	How to link it	Loaded by	At
`.COM`	as it is: **`tanren main sub`**	MSX-DOS, when its name is typed	0100h
Raw image	`/P:`, and `/O:` for its name	another program	the `/P:` address
`.BIN`	`/B`, and `/P:`	MSX-BASIC's `BLOAD`	the address in its header
ROM image	`ASEG`, and `ORG` 4000h or 8000h in the source, and `/O:` for its name	the MSX, from a cartridge	4000h or 8000h, where the MSX looks for the header

Table 26.1: The kinds of output file.

26.2 A .COM program

A `.COM` program is what TANREN makes without any option. MSX-DOS loads it at 0100h and starts it there, so it must be linked for 0100h, which is where TANREN starts the code (chapter 24):

MSX-DOS

```
A:\>tanren main sub
...
2 modules, 16 records, ends at EOF.
Wrote MAIN.COM, 0100-0125 (38 bytes), entry 0100.
```

Do not use `/P:` for a `.COM` program. The file would still be loaded at 0100h, and every address in it would be wrong.

26.3 A raw image for another address

With `/P:`, TANREN links the program for another address. The file is the same kind of file, the bytes and nothing else, but it only works at that address, and something other than MSX-DOS has to load it there: another program, for example, that reads the file into memory and calls it.

This module prints a line with `CHPUT`, the routine of the MSX's BIOS that prints a character (`BIOS.INC`), so that it works without MSX-DOS:

```
; HELLOB.AS - prints a line with the BIOS, for BLOAD from BASIC.
                include bios.inc
                include ascii.inc
                cseg
start:          ld      hl,msg
loop:           ld      a,(hl)
                or      a
                ret     z
                push    hl
                call    CHPUT
                pop     hl
                inc     hl
                jr      loop
                dseg
msg:            db      'Hello from BLOAD.',CHR_CR,CHR_LF,0
                end     start
```

Linked for C000h, it is 34 bytes:

MSX-DOS

```
A:\>tanren /p:c000 hellob /o:hellor.bin
...
1 modules, 8 records, ends at EOF.
Wrote HELLOR.BIN, C000-C021 (34 bytes), entry C000.
```

The file starts with `21 0E C0`, `ld hl,msg`: the message is at C00Eh, after the fourteen bytes of code.

26.4 A file for BLOAD: /B

MSX-BASIC loads a machine-code file with `BLOAD`. The file must start with a seven-byte *BLOAD header* that says where it goes, and `/B` makes TANREN write one. With `/B`, the output file gets `.bin` when it is not named with `/O:`:

MSX-DOS

```
A:\>tanren /b /p:c000 hellob
...
1 modules, 8 records, ends at EOF.
Wrote HELLOB.BIN, C000-C021 (34 bytes), BLOAD header, entry C000.
```

The summary line gives the addresses and the size of the program, as always. The file on the disk is 41 bytes, seven more, because of the header. Figure 26.1 shows the header of `HELLOB.BIN`:

- `FE`, which marks a machine-code file;

- the first address of the program, C000h;
- the address of its *last* byte, C021h, not the address after it;
- the address where it starts, the entry address, C000h (section 26.5).

Each address is two bytes, the low byte first, as the Z80 keeps them. After the header come the same 34 bytes as in `HELLOR.BIN`.

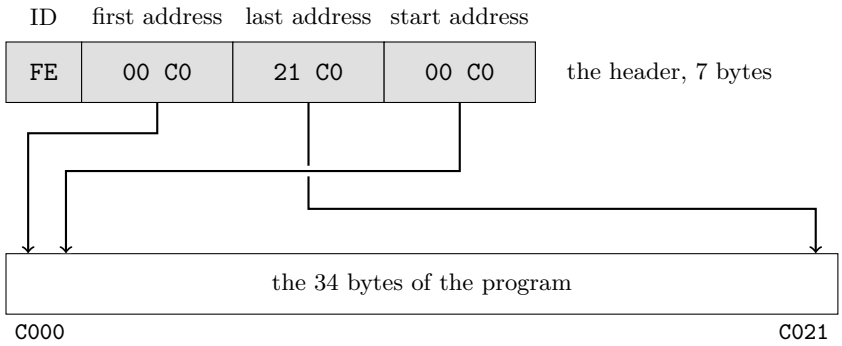


Figure 26.1: The BLOAD header of `HELLOB.BIN`, and the three addresses it holds.

In MSX-BASIC, `BLOAD` with `,R` loads the file and calls its start address, and the program returns to BASIC with `RET`:

MSX-DOS

```
Ok
bload"hellob.bin",r
Hello from BLOAD.
Ok
```

While MSX-BASIC runs, the addresses below 8000h hold its ROM, and the RAM starts at 8000h, so a `.BIN` file has to be linked for an address in that RAM, where it does not overwrite what BASIC is using. A `.BIN` linked for 0100h cannot be loaded at all. Chapter 32 is a complete example.

26.5 The entry address

The *entry address* is the address named on an `END` line (section 12.6). When more than one module names one, the first in command-line order counts, and the others are ignored. `E1.AS` and `E2.AS` each name one:

```
; E1.AS - its END names e1start.
                cseg
e1start:        nop
                ret
                end     e1start
```

```
; E2.AS - its END names e2start, which is not its first byte.
                cseg
                nop
e2start:        ret
                end     e2start
```

MSX-DOS

```
A:\>tanren /b /p:c000 e1 e2 /o:e12.bin
...
Wrote E12.BIN, C000-C003 (4 bytes), BLOAD header, entry C000.
A:\>tanren /b /p:c000 e2 e1 /o:e21.bin
...
Wrote E21.BIN, C000-C003 (4 bytes), BLOAD header, entry C001.
```

With `E1` first, the entry is `e1start`, at C000h; with `E2` first, it is `e2start`, at C001h.

When no module names one, the summary line has no `entry`:

```
; NOEND.AS - no address on its END line.
                cseg
                ld      a,1
                ret
                end
```

MSX-DOS

```
A:\>tanren noend
...
Wrote NOEND.COM, 0100-0102 (3 bytes).
```

A BLOAD header always has an entry address, so with `/B` TANREN uses the first address of the program, and the summary line says so:

MSX-DOS

```
A:\>tanren /b /p:c000 noend
...
Wrote NOEND.BIN, C000-C002 (3 bytes), BLOAD header, entry C000.
```

The entry address is used only in the BLOAD header. MSX-DOS always starts a `.COM` program at 0100h (section 21.1), and a raw image is usually entered at its

first address too. Without /B, TANREN therefore warns when the entry address is anything else, and still writes the file:

MSX-DOS

```
A:\>tanren e2 e1 /o:e21raw.bin
...
WARNING: E21RAW.BIN is entered at 0100, not at 0101.
2 modules, 12 records, ends at EOF.
Wrote E21RAW.BIN, 0100-0103 (4 bytes), entry 0101.
```

The warning is printed even with /Q. With /B there is none, as E21.BIN above shows: the header tells BLOAD where to start.

26.6 A ROM image

A ROM cartridge of 16 KB sits at 4000h to 7FFFh, and starts with a header that the MSX looks for when it starts. ROM.AS, in EXAMPLES\ROM, is a complete one (its first lines, a comment, are shortened here):

```
; ROM.AS - a 16 KB cartridge.

                include bios.inc        ; INITXT, CHPUT, and the rest

                aseg
                org     4000h

                db      "AB"            ; a cartridge, says the BIOS
                dw      init            ; INIT: called at once
                dw      0               ; STATEMENT: no CALL statements
                dw      0               ; DEVICE: no device
                dw      0               ; TEXT: no BASIC program in here
                dw      0,0,0           ; reserved, and zero

init:           call    INITXT          ; a text screen, from nothing
                ld      hl,msg
ini.lp:         ld      a,(hl)
                or      a
                jr      z,ini.end
                push    hl
                call    CHPUT
                pop     hl
                inc     hl
                jr      ini.lp
ini.end:        jr      ini.end         ; a cartridge has nowhere to go

msg:            db      "Assembled by Tatara. Linked by TANREN.",13,10,0
```

```
                org     7fffh           ; the last byte of the 16 KB...
                db      0               ;   ...so that the file is all of it

                end
```

The code is in the absolute segment, at the addresses written in the source (section 24.5), so no `/P:` is needed:

MSX-DOS

```
A:\>tanren rom /o:rom.rom
...
1 modules, 6 records, ends at EOF.
Wrote ROM.ROM, 4000-7FFF (16384 bytes).
```

A cartridge image has to be the full 16,384 bytes, and the last two lines make it so: TANREN writes everything from the first byte with content to the last, and a single byte at 7FFFh makes the gap before it part of the file, as zeros. Without those two lines, the same program makes a file of 77 bytes:

MSX-DOS

```
A:\>tanren romshort /o:romshort.rom
...
Wrote ROMSHORT.ROM, 4000-404C (77 bytes).
```

A `DS` at the end would not do it, because the memory a `DS` reserves after the last byte of content is not written to the file (section 24.1.1). Chapter 33 describes the cartridge in full.

26.7 How large a program can be

TANREN builds the program in the memory mapper, not in the memory that MSX-DOS gives to programs, and writes it to the file from there. A program can therefore use the whole address space, from 0100h to FFFFh, including the addresses where TANREN itself is running. This module has a byte at 0100h and another more than 60 KB above it:

```
; BIG.AS - one byte at the start and one near the top of memory.
                cseg
                db      1
                ds      0EF00h
                db      2
                end
```

MSX-DOS

```
A:\>tanren big
...
1 modules, 6 records, ends at EOF.
Wrote BIG.COM, 0100-F001 (61186 bytes).
```

A large program needs enough free mapper memory for its image, as well as for TANREN's tables, whose size depends on the program: how many modules, segments and names it has. When the mapper runs out, TANREN stops with `ERROR: out of mapper memory.`

A `.COM` program still has to fit in the memory MSX-DOS gives it when it runs, which ends well below FFFFh. TANREN does not check this: the limit depends on the MSX and on MSX-DOS, not on the linker.

26.8 When there is nothing to write

A program whose modules hold only `DS` has no byte with content, and TANREN writes no file:

```
; EMPTY.AS - only DS: nothing to write.
                dseg
buffer:         ds      100
                end
```

MSX-DOS

```
A:\>tanren empty
...
1 modules, 4 records, ends at EOF.
Nothing to write: no module has any content.
```

This is not an error, but no output file is made, and one of the same name from an earlier link is left as it was.

26.9 Messages

Table 26.2 lists the messages about the output file. `ERROR: the linked image would run past FFFFh.` is described in chapter 24.

Message	Meaning
`ERROR: cannot create NAME`	The output file cannot be made, for example because its directory does not exist, as with **`/o:nodir\x.com`**. The name is shown in capitals.
`ERROR: cannot write the output file - the disk may be full.`	The file was made, but writing to it failed.
`ERROR: out of mapper memory.`	The memory mapper has no room left for the program's image or for TANREN's tables.
`WARNING: NAME is entered at XXXX, not at YYYY.`	Not an error: the file, written without `/B`, will be entered at its first address, XXXX, but the `END` address is YYYY.
`Nothing to write: no module has any content.`	Not an error: no module has a byte with content, and no file is made.

Table 26.2: Messages about the output file.

CHAPTER 27

Linker error messages

27.1 What a message says 203
27.2 TANREN stops at the first error 204
27.3 Damaged object files 205
27.4 Segments 206
27.5 Messages 207

When TANREN finds a mistake, it prints one error message and stops, as TATARA does (chapter 20). This chapter describes what TANREN's messages look like, what it leaves on the disk when it stops, and the messages that the earlier chapters have not described: those about damaged object files and about segments. Appendix D lists every message, with its cause.

27.1 What a message says

A TANREN message is one line that starts with `ERROR:`. TANREN has no source line to point at, so a message names no line. When the mistake is in one object file, the message starts with that file's name, in capitals, as TATARA's messages start with the name of the source:

MSX-DOS

```
A:\>tanren main bad
...
BAD.TRO: ERROR: not a Tatara object file.
```

The other messages name a file inside the text, as in `ERROR: cannot open MAIN,SUB.TRO` (chapter 21), or name none, because the mistake belongs to no one file, as in `ERROR: /D: would put the data on top of the code.` (chapter 24).

A few print lines of their own before the error, such as the list of names never defined (chapter 25).

When a message comes also says something about the mistake. Figure 27.1 shows the stages of a link, and the mistakes TANREN can find at each one. The messages about the command line and the link file come before the banner (section 21.6); the rest come after it.

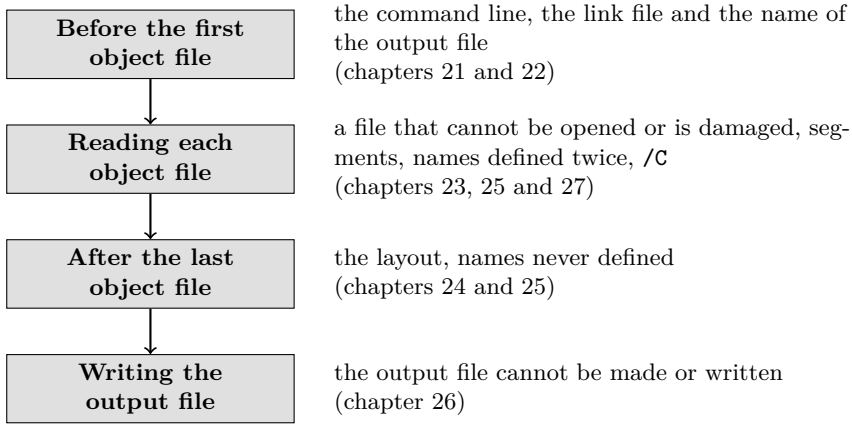


Figure 27.1: The stages of a link, and the mistakes TANREN finds at each one.

`/Q` leaves out the banner and the summary line, but never an error message.

27.2 TANREN stops at the first error

TANREN prints one message and stops. Correct the mistake and link again, and TANREN goes on to the next one, if there is one. With two damaged object files on the command line, only the first is reported:

MSX-DOS

```
A:\>tanren sub bad trunc /o:x.com
...
BAD.TRO: ERROR: not a Tatara object file.
```

The one exception is names that no module defines: TANREN lists all of them before it stops (section 25.3).

TANREN writes the output file only after it has read every object file twice, so an error stops it before the file is made. An output file of the same name from an

earlier link is left as it was. `MAIN.COM` here is the one **`tanren main sub`** wrote, and a failed link of the same name does not change it:

MSX-DOS

```
A:\>dir main.com
...
MAIN     COM        38 26-10-01  8:25
 38 bytes in 1 file   2336K free
A:\>tanren main bad
...
BAD.TRO: ERROR: not a Tatara object file.
A:\>dir main.com
...
MAIN     COM        38 26-10-01  8:25
 38 bytes in 1 file   2336K free
```

After an error, the output file on the disk is not the program on the command line, but an older one. Do not run it until TANREN has linked without an error.

> Warning. `ERROR: cannot write the output file - the disk may be full.` comes after TANREN has created the output file, while it is writing it. TANREN stops at once, and does not touch the disk again: after a disk error, anything more it did could do more damage. The output file is left on the disk, with only part of the program in it. Delete it, and make room on the disk or use another one, before linking again.

27.3 Damaged object files

An object file starts with a few bytes that mark it as a TATARA object file and give the version of its format (appendix F). TANREN checks them when it opens the file, and follows the file's records to the end. These messages mean that the file is not what TATARA makes:

MSX-DOS

```
A:\>tanren main bad
...
BAD.TRO: ERROR: not a Tatara object file.
A:\>tanren trunc sub /o:x.com
...
TRUNC.TRO: ERROR: the object file ends inside a record.
A:\>tanren oldver sub /o:x.com
...
OLDVER.TRO: ERROR: this object file was made by another version.
```

- `not a Tatara object file.` The file does not start as an object file does. `BAD.TRO` is a source file that was given the extension `.tro` by mistake.
- `the object file ends inside a record.` The file stops in the middle of its contents. `TRUNC.TRO` is the first 30 bytes of a good object file, as a file cut short by a full disk or an interrupted copy would be.
- `this object file was made by another version.` The file is a TATARA object file, but in a format this TANREN does not read. Each version of TANREN reads the object files of the TATARA released with it.

In each case, assemble the module again with this version of TATARA, and link again.

27.3.1 Bytes after the end

The last record of an object file marks its end. When a file has anything after that record, TANREN names the file, and goes on:

MSX-DOS

```
A:\>tanren tail sub /o:tail.com
...
TAIL.TRO: bytes follow the END record.
2 modules, 16 records, AND BYTES AFTER THEM.
Wrote TAIL.COM, 0100-0125 (38 bytes), entry 0100.
```

This is not an error: the bytes after the end are not read, and the output file is written. But TATARA does not make such a file, so something else has changed it since it was assembled. Assemble that module again. `/Q` leaves out both lines.

27.4 Segments

A segment named in several modules is one segment (chapter 24), so it must be the same kind of segment in every module. `SEGA.AS` makes `shared` a code segment, and `SEGB.AS` a data segment:

```
; SEGA.AS - shared is a code segment here.
                public  ina
                cseg    shared
ina:            nop
                end
```

```
; SEGB.AS - and a data segment here.
                public  inb
                dseg    shared
inb:            db      0
                end
```

MSX-DOS

```
A:\>tanren sega segb /o:x.com
...
SEGB.TRO: ERROR: segment shared is declared differently.
```

The file named is the one where the second, different declaration was found: with **`tanren segb sega`**, it is `SEGA.TRO`.

27.4.1 The limits of one module

TANREN keeps up to 16 segments and 16 groups for each module. TATARA accepts more, so a module with too many is found only when it is linked:

MSX-DOS

```
A:\>tanren seg16 /o:x.com
...
SEG16.TRO: ERROR: too many segments or groups in one module.
```

`SEG16.AS` has the default code segment and fifteen named ones. Every object file also has the default data segment, even when nothing is in it, so this module has seventeen segments. A module can therefore have up to fourteen named segments, besides the default code and data segments. The groups of transient segments (section 10.6) are counted separately, up to sixteen in one module.

A program that needs more segments or groups can spread them over several modules: the limits apply to each module, not to the program.

27.5 Messages

Table 27.1 lists the messages described in this chapter. The others are described in chapters 21 to 26, and appendix D lists them all.

Message	Meaning
`FILE: ERROR: not a Tatara object file.`	The file does not start as an object file does.
`FILE: ERROR: the object file ends inside a record.`	The file stops in the middle of its contents.
`FILE: ERROR: this object file was made by another version.`	The file's format is not the one this TANREN reads.
`FILE: ERROR: segment NAME is declared differently.`	The segment is a different kind in this file and an earlier one.
`FILE: ERROR: too many segments or groups in one module.`	The module has more than 16 segments or more than 16 groups.
`FILE: bytes follow the END record.`	Not an error: the file has bytes after its last record.
`ERROR: cannot write the output file - the disk may be full.`	Writing the output file failed. Part of it is left on the disk.

Table 27.1: The messages described in this chapter.

PART IV

Worked guides

28 A program in several modules 211

29 Scratch variables shared through a transient segment 219

30 Macros and the listing modes 229

31 Long symbol names and case-sensitive names 239

32 A binary for MSX-BASIC's BLOAD 247

33 A 16 KB ROM cartridge 255

34 A project in several directories 263

35 Building TATARA and TANREN from their own sources 271

A Directive reference 279

CHAPTER 28

A program in several modules

28.1 The example 211
28.2 The two modules 212
28.3 The link file 213
28.4 Building and running it 214
28.5 What TANREN did 215
28.6 Things to try 216

This chapter builds `TWOMOD`, the example in `A:\TATARA\EXAMPLES\TWOMOD`: a program of two modules, linked from a link file. It goes through the files one at a time, builds the program, looks at what TANREN made of it, and then changes it in three ways to show what goes wrong, and why. The reference chapters named along the way describe each part in full.

28.1 The example

`TWOMOD` prints two lines and ends:

MSX-DOS

```
A:\TATARA\EXAMPLES\TWOMOD>twomod
Two modules, one program.
The linker joined them.
```

The printing is done by a routine, `putstr`, that is in a module of its own. The example has four files:

- `MAIN.AS`, the module where the program starts;

- `PUTSTR.AS`, the module with `putstr`;
- `TWOMOD.LNK`, the link file that names the two modules;
- `BUILD.BAT`, a batch file that assembles and links them.

28.2 The two modules

`MAIN.AS` calls `putstr` twice, once for each line, and then ends the program:

```
; MAIN.AS - one of two modules.
;
; It knows that putstr exists. It does not know where putstr is, and
; it does not need to: EXTRN says the name is somebody else's, and the
; linker fills the address in.

                extrn   putstr          ; defined in PUTSTR.AS

                include msxdos.inc      ; _TERM0 and "system"
                include ascii.inc       ; CHR_CR and CHR_LF

                cseg

start:          ld      hl,msg1
                call    putstr
                ld      hl,msg2
                call    putstr
                system  _TERM0

msg1:           db      "Two modules, one program.",CHR_CR,CHR_LF,0
msg2:           db      "The linker joined them.",CHR_CR,CHR_LF,0

                end     start
```

`extrn putstr` tells TATARA that `putstr` is defined in some other module (section 12.5). TATARA does not know where, so it leaves the address in each `call putstr` to be filled in by TANREN. The `END` line names `start`, the first instruction.

`PUTSTR.AS` defines `putstr`, which prints the characters of a string one at a time until it reaches a zero byte:

```
; PUTSTR.AS - the other module.
;
; PUBLIC is what makes the name visible outside this file. Without it
; the name would still assemble here and the link of MAIN.AS would
; stop with an undefined symbol - which is worth trying once.

                public  putstr

                include msxdos.inc      ; BDOS, _CONOUT and "system"

                cseg

; putstr - print a string.
;
;   HL is pushed around the BDOS call because MSX-DOS promises nothing
;   about the registers it hands back.
;
; Input:        HL -> the string, ending in a zero byte
; Output:       it is printed
; Modifies:     AF, BC, DE, HL

putstr:         ld      a,(hl)
                or      a
                ret     z
                push    hl
                ld      e,a
                system  _CONOUT
                pop     hl
                inc     hl
                jr      putstr

                end
```

`public putstr` offers the name to the other modules. Neither module knows anything else about the other: `MAIN.AS` knows only the name `putstr`, and `PUTSTR.AS` does not know who calls it. Each is assembled on its own, into an object file of its own.

28.3 The link file

`TWOMOD.LNK` holds the rest of the link (chapter 22):

```
; TWOMOD.LNK - the link, in a file instead of on the command line.
;
; TANREN reads this when it is given @TWOMOD, and the words in it are
; the words it would have read from the command line. A ; starts a
; comment and the rest of the line is ignored.
;
; MSX-DOS gives a command line 127 characters and cuts a longer one
```

```
; without saying so. That is the whole reason this exists: the
; assembler is itself linked from a file of nineteen module names.

/o:twomod.com

main.tro putstr.tro
```

After the comment come two lines of words: `/o:twomod.com`, the name of the output file, and the two object files. `MAIN.TRO` is named first, because MSX-DOS starts a `.COM` program at 0100h, and TANREN puts the code of the first module there (section 21.1).

28.4 Building and running it

`BUILD.BAT` makes the program:

```
rem BUILD.BAT - EXAMPLE: TWOMOD
rem
rem Two modules, one program. MAIN.AS declares putstr EXTRN and
rem PUTSTR.AS declares it PUBLIC; the linker matches the two.
rem
rem The link itself is in TWOMOD.LNK rather than on this line. @NAME
rem reads a file of words, and .LNK is assumed when the name has no
rem extension. Everything in it could have been typed here instead.

rem The headers are in A:\TATARA\INCLUDE and are not copied here.
rem TATARA is rule 3 of the include search - see the DIRS example.
rem CHANGE THE PATH BELOW if you put the tree somewhere else.

set TATARA=a:\tatara\include

echo === Assembling both modules
tatara /q main.as main.tro
tatara /q putstr.as putstr.tro

echo === Linking from TWOMOD.LNK
tanren /q @twomod

echo === Done. Type TWOMOD to run it.

set TATARA=
```

It sets `TATARA`, so that TATARA finds `MSXDOS.INC` and `ASCII.INC` (section 13.3), assembles each module into its object file, links them with **`@twomod`**, and clears `TATARA` again. Figure 28.1 shows the files it reads and makes.

Run it from the example's directory. `/Q` keeps TATARA and TANREN quiet, so

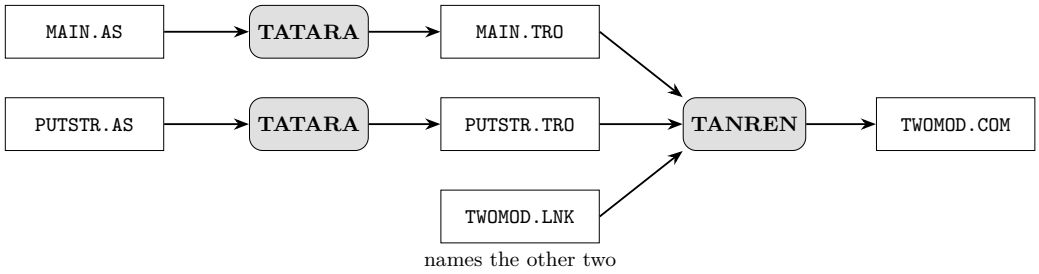


Figure 28.1: How `BUILD.BAT` makes `TWOMOD.COM`.

only the batch file's own lines are printed:

MSX-DOS

```
A:\TATARA\EXAMPLES\TWOMOD>build
=== Assembling both modules
=== Linking from TWOMOD.LNK
=== Done. Type TWOMOD to run it.
A:\TATARA\EXAMPLES\TWOMOD>twomod
Two modules, one program.
The linker joined them.
```

The same link without `/Q` shows TANREN's summary. It needs `TATARA` only for assembling, so this link can be typed as it is:

MSX-DOS

```
A:\TATARA\EXAMPLES\TWOMOD>tanren @twomod
Tatara MSX Linker v1.2.0
Copyright (C) 2026 Javier Lavandeira
https://tatara.tools

2 modules, 14 records, ends at EOF.
Wrote TWOMOD.COM, 0100-0154 (85 bytes), entry 0100.
```

28.5 What TANREN did

`/M` shows where TANREN put everything (section 25.7):

MSX-DOS

```
A:\TATARA\EXAMPLES\TWOMOD>tanren /m @twomod /o:tm.com
...
Modules: 2
Groups:
Segments:
  flags 00 group FF base 0100 size 0055 end 0154  C
  flags 01 group FF base 0155 size 0000 end -----  D
Symbols:
  DR 0147 putstr
2 modules, 14 records, ends at EOF.
Wrote TM.COM, 0100-0154 (85 bytes), entry 0100.
```

Both modules put everything in the default code segment, so the program is one code segment of 55h bytes, from 0100h to 0154h: the 71 bytes of `MAIN` first, then the 14 of `PUTSTR`. The data segment is empty. `putstr` is defined (`D`) by `PUTSTR`, referenced (`R`) by `MAIN`, and at 0147h, the first byte after `MAIN`. TANREN wrote 0147h into both `call putstr` instructions (figure 28.2).

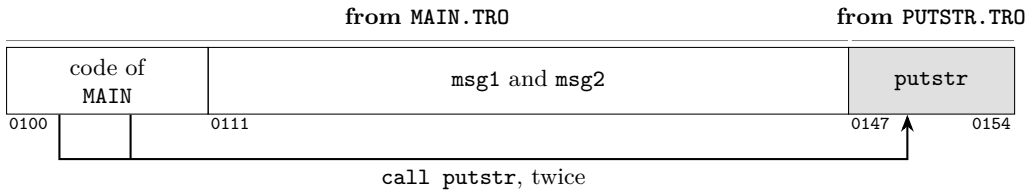


Figure 28.2: `TWOMOD.COM` in memory. Both calls in `MAIN` go to `putstr`, at 0147h.

28.6 Things to try

Each change below shows one of the mistakes that TANREN finds. The files it uses are not part of the example: make them as copies of its files, in the same directory. The assembling needs `TATARA`, as in `BUILD.BAT`:

```
set tatara=a:\tatara\include
```

28.6.1 Leave out PUBLIC

`NOPUB.AS` is `PUTSTR.AS` without its `public putstr` line. It still assembles: `putstr` is an ordinary label in it. But no module offers the name any more, and the link fails (section 25.3):

MSX-DOS

```
A:\TATARA\EXAMPLES\TWOMOD>tatara /q nopub.as nopub.tro
A:\TATARA\EXAMPLES\TWOMOD>tanren main nopub /o:np.com
...
Never defined:
  putstr
ERROR: the symbols above were never defined.
```

28.6.2 Name the modules the other way round

`SWAP.LNK` is `TWOMOD.LNK` with `putstr.tro  main.tro` on its last line, and `/o:swap.com`. It links without an error, but with a warning:

MSX-DOS

```
A:\TATARA\EXAMPLES\TWOMOD>tanren /m @swap
...
Symbols:
  DR 0100 putstr
WARNING: SWAP.COM is entered at 0100, not at 010E.
2 modules, 14 records, ends at EOF.
Wrote SWAP.COM, 0100-0154 (85 bytes), entry 010E.
```

But `putstr` is now at 0100h, and `start` at 010Eh. MSX-DOS starts `SWAP.COM` at 0100h, so it runs `putstr` with whatever `HL` happens to hold, and never reaches the code of `MAIN`. Do not run it. The entry address, 010Eh, is right, but a `.COM` program cannot start there, and TANREN says so in its warning (section 26.5).

28.6.3 Assemble one module with /C

`MAIN.AS` assembled with `/C`, and `PUTSTR.AS` without it, cannot be linked together (section 25.5):

MSX-DOS

```
A:\TATARA\EXAMPLES\TWOMOD>tatara /q /c main.as mainc.tro
A:\TATARA\EXAMPLES\TWOMOD>tanren mainc putstr /o:mix.com
...
ERROR: one module was assembled /C and another was not.
```

CHAPTER 29

Scratch variables shared through a transient segment

29.1 The example 219
29.2 The source 220
29.3 Building it 221
29.4 Reading the symbol table 222
29.5 Reading the maps 223
29.6 Things to try 224

A program often has variables that only one part of it uses, and only while that part runs: a buffer for a line being read, another for a line being written. When the two parts never run at the same time, the two buffers can share the same memory. A transient segment (section 10.6) does that, and this chapter follows it through `LAYOUT`, the example in `A:\TATARA\EXAMPLES\LAYOUT`: the source, the symbol table TATARA makes of it, and the maps TANREN makes when it links it at two different addresses.

29.1 The example

`LAYOUT` is one module, `LAYOUT.AS`, and a `BUILD.BAT`. The program itself only prints a line:

MSX-DOS

```
A:\TATARA\EXAMPLES\LAYOUT>layout
Code here, data elsewhere.
```

What the example is for is where its bytes go. It has code, two plain variables, and a transient segment, SCRATCH, with two groups of variables that share the same memory. BUILD.BAT writes the symbol table and two maps into files, so that they can be read after the build.

29.2 The source

```
; LAYOUT.AS - where the bytes go.
;
; Three kinds of relocatable space, and the options that place them:
;
;   CSEG                        code            TANREN /P:<addr>
;   DSEG                        data            TANREN /D:<addr>
;   DSEG <name>,TRANSIENT       overlaid data   one GROUP at a time
;
; A SEGMENT MAY BE NAMED, and a named one keeps its own counter: two
; CSEG MUSIC lines in different files contribute to one segment.
;
; A TRANSIENT DSEG is space that is used by one part of a program at a
; time. Every GROUP in it starts at the same address, so the segment is
; as large as its largest group and not as large as their sum. Labels
; in two different groups cannot be subtracted from each other,
; because nothing says which of them is in memory.
;
; The absolute kind, ASEG, is in the BINARY and ROM examples - a fixed
; address in a .COM would make the file span everything from 0100h up
; to it.

                include msxdos.inc      ; _STROUT, _TERM0 and "system"
                include ascii.inc       ; CHR_CR and CHR_LF

                cseg

start:          ld      hl,runs         ; a byte in the plain DSEG
                inc     (hl)
                ld      de,msg
                system  _STROUT
                system  _TERM0

msg:            db      "Code here, data elsewhere.",CHR_CR,CHR_LF,"$"

                dseg                    ; the plain data segment: the
runs:           ds      1               ;   linker puts it after the code
lastkey:        ds      1               ;   unless /D: says where
```

```
                dseg    SCRATCH,TRANSIENT
                group   READING         ; 66 bytes...
inbuf:          ds      64
inlen:          ds      2
                group   WRITING         ; ...and 34, at the same address,
outbuf:         ds      32              ;   so SCRATCH is 66 bytes and not
outlen:         ds      2               ;   100

                end     start
```

The source has three parts, one for each kind of segment:

- **The code**, in the default code segment: it adds one to `runs`, prints `msg` with the MSX-DOS function `_STROUT`, and ends.
- **Two plain variables**, `runs` and `lastkey`, in the default data segment. Each has a byte of its own.
- **The transient segment** `SCRATCH`, with two groups. `READING` holds a buffer for a line being read and its length, 66 bytes. `WRITING` holds a buffer for a line being written and its length, 34 bytes. Both groups start at the beginning of `SCRATCH`, so `inbuf` and `outbuf` are the same address, and `SCRATCH` takes 66 bytes, not 100.

The program does not use the two groups; they are there to show where they go. A real program would use `READING` while it reads a line, and `WRITING` while it writes one, and never both at once: writing to `outbuf` overwrites `inbuf` (section 10.6).

29.3 Building it

```
rem BUILD.BAT - EXAMPLE: LAYOUT
rem
rem This example is about the two maps it writes, not about the
rem program, which only prints a line.
rem
rem   TATARA /S   the symbol table: every name, by segment, with its
rem               offset inside that segment
rem   TANREN /M   the segment and group tables: where each segment
rem               ended up, how big it is, and which groups share it
rem
rem It links twice. The first is an ordinary .COM at 0100h. The second
rem puts the code at 8000h and the data at C000h - a real MSX layout,
rem code in the top pages with its variables above it, and not
rem something MSX-DOS can run as a command, so it goes to a .BIN that
```

```
rem nobody is asked to run.

rem The headers are in A:\TATARA\INCLUDE and are not copied here.
rem TATARA is rule 3 of the include search - see the DIRS example.
rem CHANGE THE PATH BELOW if you put the tree somewhere else.

set TATARA=a:\tatara\include

echo === Assembling, keeping the symbol table
tatara /q /s layout.as layout.tro > layout.sym

echo === Linking at 0100h, keeping the map
tanren /q /m /o:layout.com layout.tro > layout.map

echo === Linking again, code at 8000h and data at C000h
tanren /q /m /o:layout8.bin /p:8000 /d:c000 layout.tro > layout8.map

echo === Done. Read LAYOUT.SYM, LAYOUT.MAP and LAYOUT8.MAP.

set TATARA=
```

Each command sends what it prints to a file with >. With `/Q`, TATARA and TANREN print no banner and no summary, so the files hold only the symbol table (`/S`, chapter 19) and the maps (`/M`, section 25.7). The second link puts the code at 8000h and the data at C000h (section 24.3). Its output file, `LAYOUT8.BIN`, is not a program that MSX-DOS can run, and is not meant to be run: it is there for its map.

MSX-DOS

```
A:\TATARA\EXAMPLES\LAYOUT>build
=== Assembling, keeping the symbol table
=== Linking at 0100h, keeping the map
=== Linking again, code at 8000h and data at C000h
=== Done. Read LAYOUT.SYM, LAYOUT.MAP and LAYOUT8.MAP.
```

29.4 Reading the symbol table

`LAYOUT.SYM` starts with the absolute section, which holds every `EQU` of `MSXDOS.INC` and `ASCII.INC`, more than a hundred lines, cut here. The sections for the program's own names follow it:

MSX-DOS

```
A:\TATARA\EXAMPLES\LAYOUT>type layout.sym
ASEG - absolute

0000h             CHR_NUL
0000h             _TERM0
...
```

```
007Fh             CHR_DEL

CSEG - default code segment, 002Eh bytes

0000h             start
0011h             msg

DSEG - default data segment, 0002h bytes

0000h             runs
0001h             lastkey

SCRATCH - named data segment, transient, 0000h bytes

(no symbols)

SCRATCH - named data segment, transient, group READING, 0042h bytes

0000h             inbuf
0040h             inlen

SCRATCH - named data segment, transient, group WRITING, 0022h bytes

0000h             outbuf
0020h             outlen
```

Each value is an offset inside its section. `SCRATCH` has three sections: one for the part of the segment outside every group, which is empty here, and one for each group. `inbuf` and `outbuf` are both at 0000h, each in its own group: the symbol table cannot say yet that they will be at the same address, because only TANREN places the groups.

29.5 Reading the maps

`LAYOUT.MAP` is the map of the `.COM` program:

MSX-DOS

```
A:\TATARA\EXAMPLES\LAYOUT>type layout.map
Modules: 1
Groups:
  READING
  WRITING
Segments:
  flags 00 group FF base 0100 size 002E end 012D  C
  flags 01 group FF base 012E size 0002 end 012F  D
  flags 03 group 00 base 0130 size 0042 end 0171 SCRATCH
  flags 03 group 01 base 0130 size 0022 end 0151 SCRATCH
```

```
Symbols:
```

The code starts at 0100h and takes 2Eh bytes; the two plain variables follow it at 012Eh; and `SCRATCH` comes last, with a line for each group. Both lines have the same base, 0130h: `READING` runs to 0171h, and `WRITING`, the smaller, to 0151h. `Symbols:` is empty, because the module declares no public or external names.

`LAYOUT8.MAP` is the map of the second link:

MSX-DOS

```
A:\TATARA\EXAMPLES\LAYOUT>type layout8.map
Modules: 1
Groups:
  READING
  WRITING
Segments:
  flags 00 group FF base 8000 size 002E end 802D  C
  flags 01 group FF base C000 size 0002 end C001  D
  flags 03 group 00 base C002 size 0042 end C043 SCRATCH
  flags 03 group 01 base C002 size 0022 end C023 SCRATCH
Symbols:
```

The segments are the same, and so are their sizes; only the addresses change. The code is at 8000h, and all the data, the plain variables and `SCRATCH`, at C000h. Figure 29.1 shows both links.

Neither file holds `SCRATCH`: its bytes are reserved with `DS`, which puts nothing in the file (section 24.1.1). `LAYOUT.COM` is 46 bytes, from 0100h to 012Dh.

29.6 Things to try

The files below are not part of the example: make them in its directory, with `TATARA` set as in `BUILD.BAT`.

29.6.1 Subtract a label in one group from a label in another

```
; SUBGRP.AS - a label from each group of one transient segment,
; in one expression.
                dseg    SCRATCH,TRANSIENT
                group   READING
inbuf:          ds      64
                group   WRITING
outbuf:         ds      32
                cseg
```

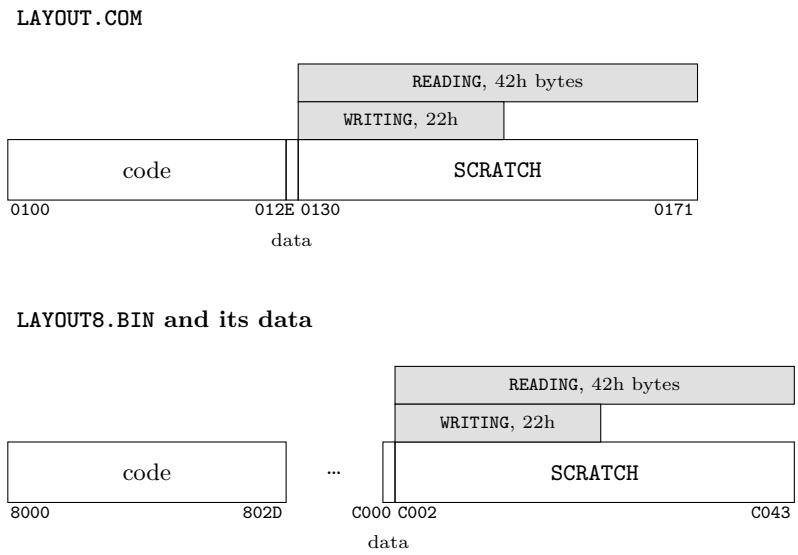


Figure 29.1: `LAYOUT` linked at 0100h, and with the code at 8000h and the data at C000h. In both, the two groups of `SCRATCH` start at the same address.

```
                ld      hl,outbuf-inbuf
                end
```

MSX-DOS

```
A:\TATARA\EXAMPLES\LAYOUT>tatara subgrp.as subgrp.tro
...
SUBGRP.AS(9): ERROR: relocation error - a segment-relative value is not allowed
    here.
```

Both labels are at offset 0000h, but in different groups, and TATARA treats them as two unrelated addresses (section 10.6).

29.6.2 Add a group in another module

`PARSE.AS` adds a third group, `PARSING`, of 80 bytes:

```
; PARSE.AS - a third group of SCRATCH, larger than the other two.
                public  tokens
                dseg    SCRATCH,TRANSIENT
                group   PARSING
tokens:         ds      80
                end
```

MSX-DOS

```
A:\TATARA\EXAMPLES\LAYOUT>tatara /q parse.as parse.tro
A:\TATARA\EXAMPLES\LAYOUT>tanren /m /o:lp.com layout parse
...
Groups:
  READING
  WRITING
  PARSING
Segments:
  flags 00 group FF base 0100 size 002E end 012D  C
  flags 01 group FF base 012E size 0002 end 012F  D
  flags 03 group 00 base 0130 size 0042 end 0171 SCRATCH
  flags 03 group 01 base 0130 size 0022 end 0151 SCRATCH
  flags 03 group 02 base 0130 size 0050 end 017F SCRATCH
Symbols:
  D- 0130 tokens
2 modules, 20 records, ends at EOF.
Wrote LP.COM, 0100-012D (46 bytes), entry 0100.
```

`PARSING` starts at 0130h too, and, being the largest group, makes `SCRATCH` 80 bytes long, to 017Fh. The groups of a transient segment overlay each other across modules (section 24.4).

29.6.3 Add to a group from another module

`READ2.AS` adds a variable to `READING`, the group that is already in `LAYOUT.AS`:

```
; READ2.AS - more variables for the group READING, from another module.
                public  linenum
                dseg    SCRATCH,TRANSIENT
                group   READING
linenum:        ds      2
                end
```

MSX-DOS

```
A:\TATARA\EXAMPLES\LAYOUT>tatara /q read2.as read2.tro
A:\TATARA\EXAMPLES\LAYOUT>tanren /m /o:lr.com layout read2
...
Segments:
  flags 00 group FF base 0100 size 002E end 012D  C
  flags 01 group FF base 012E size 0002 end 012F  D
  flags 03 group 00 base 0130 size 0044 end 0173 SCRATCH
  flags 03 group 01 base 0130 size 0022 end 0151 SCRATCH
Symbols:
  D- 0172 linenum
2 modules, 20 records, ends at EOF.
Wrote LR.COM, 0100-012D (46 bytes), entry 0100.
```

A group of the same name in another module is the same group. Its variables follow the ones before it, as in any segment: `linenum` comes after `inbuf` and `inlen`, at 0172h, and `READING` grows to 44h bytes. This is how the parts of a large program share scratch memory: each module that works with lines being read adds its variables to `READING`, and none of them needs to know about the others.

29.6.4 Forget TRANSIENT

`PLAIN.AS` declares `SCRATCH` without `TRANSIENT`:

```
; PLAIN.AS - SCRATCH again, but not transient.
                public  plainv
                dseg    SCRATCH
plainv:         ds      10
                end
```

MSX-DOS

```
A:\TATARA\EXAMPLES\LAYOUT>tatara /q plain.as plain.tro
A:\TATARA\EXAMPLES\LAYOUT>tanren /m /o:lpl.com layout plain
...
PLAIN.TRO: ERROR: segment SCRATCH is declared differently.
```

A segment has to be declared the same way in every module (section 27.4).

CHAPTER 30

Macros and the listing modes

30.1 The example 229
30.2 The source 230
30.3 Building and running it 231
30.4 The three listing modes 233
30.5 Things to try 236

A macro turns one line of the source into several, and a repeat block turns a few lines into many. The listing (chapter 18) shows what they turned into, and how much of it is up to the source. This chapter follows both through `MACROS`, the example in `A:\TATARA\EXAMPLES\MACROS`: the macros and repeat blocks in its source, the program they make, and the same lines in the three listing modes.

30.1 The example

`MACROS` prints two lines, with a pause after each:

MSX-DOS

```
A:\TATARA\EXAMPLES\MACROS>macros
A macro is a line that becomes many.
LOCAL is why DELAY can be used twice.
```

The example has three files: the source, `MACROS.AS`; a second source of two lines, `LALL.AS`, that lists the first in full; and `BUILD.BAT`, which makes the program and two listings of it.

30.2 The source

```
; MACROS.AS - a line that becomes many.
;
; MACRO defines one, ENDM ends the definition, and the name is then
; used like a mnemonic. LOCAL, REPT and IRP are below.

                include msxdos.inc      ; _STROUT, _TERM0 and "system"
                include ascii.inc       ; CHR_CR and CHR_LF

; PRINT - three instructions from one line. The parameter is
; substituted wherever the name appears in the body.

print           macro   addr
                ld      de,addr
                system  _STROUT ; a macro inside a macro
                endm

; DELAY - and why LOCAL exists. The body has a label in it, and a
; macro used twice would define that label twice. LOCAL makes a fresh
; name for each expansion, so this may be used as often as you like.

delay           macro   n
                local   loop
                ld      bc,n
loop:           dec     bc
                ld      a,b
                or      c
                jr      nz,loop
                endm

                cseg

start:          print   msg1
                delay   20000
                print   msg2
                delay   20000
                system  _TERM0

msg1:           db      "A macro is a line that becomes many.",CHR_CR,CHR_LF,"$"
msg2:           db      "LOCAL is why DELAY can be used twice.",CHR_CR,CHR_LF,"$"

; REPT repeats a body a counted number of times, with no parameter.
; Neither it nor IRP takes a label: the body is what produces the
; bytes, and a label belongs on a line inside it.

                rept    8
                db      0ffh
                endm

; IRP repeats it once for each item in the list, and IRPC once for
; each character of a word. INSIDE A STRING the parameter is only
; substituted if an ampersand is put in front of it, which is how M80
; tells a parameter from two letters that happen to match.

                irp     n,<1,2,4,8>
                db      n
                endm
```

```
                irpc    c,TATARA
                db      '&c'
                endm
                db      0

                end     start
```

The source has two macros, a program that uses them, and three repeat blocks:

- `print` loads the address of a string into `DE` and prints it with the MSX-DOS function `_STROUT`. It does the printing with `system`, which is itself a macro, from `MSXDOS.INC`: a macro can use another (section 16.7).
- `delay` counts `BC` down from its argument to 0. Its loop needs a label, and a label can be defined only once, so `local loop` gives the label a new name each time the macro is used (section 16.5).
- **The program**, at `start`, uses each macro twice and ends with `system _TERM0`. Five lines of source become eighteen instructions.
- **The repeat blocks** add bytes after the program that it never uses; they are there for the listing. `REPT` makes eight bytes of FFh, `IRP` the bytes 1, 2, 4 and 8, and `IRPC` the letters of `TATARA`, one for each character. `'&c'` is the character: in a string, a parameter is replaced only when `&` comes before it (chapter 17, section 16.3).

`LALL.AS` is the same program, with one line before it:

```
; LALL.AS - the same program, listed in full.
;
; .LALL asks for every line of every macro expansion, including the
; ones that produce no bytes. The default is .XALL, which keeps only
; the lines that emitted something. Nothing else here differs from
; MACROS.AS, so the two listings differ only in what they show.

                .lall
                include macros.as
```

30.3 Building and running it

```
rem BUILD.BAT - EXAMPLE: MACROS
rem
rem The listing is the example here. /L writes one, and the file it
rem goes to is named THIRD on the command line, after the source and
rem the object.
rem
rem A listing shows the bytes a line produced, and a line that came
rem out of a macro is marked with a +. HOW MUCH of an expansion is
rem listed is a choice, and the source makes it three times:
rem
rem   .LALL   every line of every expansion
rem   .XALL   only the lines that produced bytes (the default)
rem   .SALL   none of them, only the call
rem
rem It is assembled twice, to two listings, so the two can be read
rem side by side.

rem The headers are in A:\TATARA\INCLUDE and are not copied here.
rem TATARA is rule 3 of the include search - see the DIRS example.
rem CHANGE THE PATH BELOW if you put the tree somewhere else.

set TATARA=a:\tatara\include

echo === Assembling, listing under the default XALL
tatara /q /l macros.as macros.tro macxall.lst

echo === Assembling again, listing everything
tatara /q /l lall.as lall.tro maclall.lst

echo === Linking
tanren /q /o:macros.com macros.tro

echo === Done. Compare MACXALL.LST with MACLALL.LST, then type MACROS.

set TATARA=
```

`/L` writes a listing to the file named third on the command line (section 18.1). `MACROS.AS` is assembled into `MACXALL.LST`, and `LALL.AS` into `MACLALL.LST`; only `MACROS.TRO` is linked.

```
A:\TATARA\EXAMPLES\MACROS>build
=== Assembling, listing under the default XALL
=== Assembling again, listing everything
=== Linking
=== Done. Compare MACXALL.LST with MACLALL.LST, then type MACROS.
A:\TATARA\EXAMPLES\MACROS>macros
A macro is a line that becomes many.
LOCAL is why DELAY can be used twice.
```

The program is 135 bytes long, from 0100h to 0186h.

30.4 The three listing modes

Each listing is about 33 KB. Both include files are listed, line by line, before the program: the program's own lines start at line 339. `.XLIST` and `.LIST` around the `INCLUDE` lines would leave them out (section 18.3). The examples below are the program's part of each listing.

30.4.1 .XALL, the default

`MACXALL.LST` lists the lines of each expansion that produce bytes, marked with +:

MSX-DOS

```
0000'                          start:          print   msg1
0000'   11 25 00        +                     ld      de,msg1
0003'   0E 09           +                     ld      c,_STROUT
0005'   CD 05 00        +                     call    BDOS
                                               delay   20000
0008'   01 20 4E        +                     ld      bc,20000
000B'   0B              +     ??0000:         dec     bc
000C'   78              +                     ld      a,b
000D'   B1              +                     or      c
000E'   20 FB           +                     jr      nz,??0000
                                               print   msg2
0010'   11 4C 00        +                     ld      de,msg2
0013'   0E 09           +                     ld      c,_STROUT
0015'   CD 05 00        +                     call    BDOS
                                               delay   20000
0018'   01 20 4E        +                     ld      bc,20000
001B'   0B              +     ??0001:         dec     bc
001C'   78              +                     ld      a,b
001D'   B1              +                     or      c
001E'   20 FB           +                     jr      nz,??0001
                                               system  _TERM0
0020'   0E 00           +                     ld      c,_TERM0
0022'   CD 05 00        +                     call    BDOS
```

Two things show here that the source does not. The label of `delay`'s loop is `??0000` the first time and `??0001` the second: those are the names `LOCAL` made (section 16.5). And the `system _STROUT` line of `print` is not listed, although its two instructions are: the line itself produces no bytes, the lines of its expansion do.

The repeat blocks are listed the same way: the block as written, and then each line it produced:

MSX-DOS

```
                                        irpc    c,TATARA
                                        db      '&c'
                                        endm
0080'   54              +               db      'T'
0081'   41              +               db      'A'
0082'   54              +               db      'T'
0083'   41              +               db      'A'
0084'   52              +               db      'R'
0085'   41              +               db      'A'
0086'   00                              db      0
```

30.4.2 .LALL

`MACLALL.LST` lists every line of every expansion. The only difference in this program is the line that uses `system`, listed without an address, because it produces no bytes of its own:

MSX-DOS

```
0000'                           start:  print   msg1
0000'   11 25 00        +               ld      de,msg1
                        +               system  _STROUT ; a macro
  inside a macro
0003'   0E 09           +               ld      c,_STROUT
0005'   CD 05 00        +               call    BDOS
```

A macro with a conditional in it shows more: under `.LALL`, the lines the conditional skipped are listed too (section 18.4).

30.4.3 .SALL

The example does not use the third mode. `SALL.AS`, like `LALL.AS`, is one line and the `INCLUDE`:

```
; SALL.AS - the same program, with no expansion listed.

                .sall
                include macros.as
```

MSX-DOS

```
A:\TATARA\EXAMPLES\MACROS>tatara /l sall.as sall.tro macsall.lst
...
ended at MACROS.AS(63)
```

`MACSALL.LST` lists no line of any expansion. Each line that uses a macro is marked with `+`, to show that there is more to it than the listing shows:

MSX-DOS

```
 0000'                  +     start:          print   msg1
                        +                     delay   20000
                        +                     print   msg2
                        +                     delay   20000
                        +                     system  _TERM0
```

The repeat blocks are listed as they are written, and none of the bytes they produce appear anywhere in the listing.

Figure 30.1 puts the three side by side for one call.

.XALL (the default)

```
0000'                   start:  print  msg1
0000'  11 25 00  +              ld     de,msg1
0003'  0E 09     +              ld     c,_STROUT
0005'  CD 05 00  +              call   BDOS
```

the lines that produce bytes

.LALL

```
0000'                   start:  print   msg1
0000'  11 25 00  +              ld      de,msg1
                 +              system  _STROUT
0003'  0E 09     +              ld      c,_STROUT
0005'  CD 05 00  +              call    BDOS
```

every line, even the use of `system` (shaded), which produces no bytes itself

.SALL

```
0000'    +  start:  print  msg1
```

only the line of the call, marked with `+`

Figure 30.1: What each mode lists of the call `print msg1`.

`.XALL` is the usual choice: every byte is in the listing, next to the line that made it. `.LALL` is for finding out what a macro does, line by line, including its conditionals. `.SALL` is for a listing that reads like the source, once the macros are trusted. The three can be mixed in one source: each applies from its line to the next of the

three.

30.5 Things to try

The files below are not part of the example: make them in its directory, with `TATARA` set as in `BUILD.BAT`.

30.5.1 Leave out LOCAL

`NOLOCAL.AS` is `MACROS.AS` without the line `local loop`. The first use of `delay` defines `loop`; the second defines it again:

MSX-DOS

```
A:\TATARA\EXAMPLES\MACROS>tatara nolocal.as nolocal.tro
...
NOLOCAL.AS(23): ERROR: this name already has a value.
    in delay, called from NOLOCAL.AS(34)
```

Line 23 is `loop:` in the definition of `delay`, and line 34 is the second `delay 20000` (section 20.3).

30.5.2 Leave out the &

`NOAMP.AS` is `MACROS.AS` with `db 'c'` in place of `db '&c'` in the `IRPC`. It assembles without an error, but inside the string, `c` is no longer the parameter; it is the letter:

MSX-DOS

```
A:\TATARA\EXAMPLES\MACROS>tatara /l noamp.as noamp.tro noamp.lst
...
ended at NOAMP.AS(63)
```

MSX-DOS

```
                                        irpc    c,TATARA
                                        db      'c'
                                        endm
  0080'   63            +               db      'c'
  0081'   63            +               db      'c'
  0082'   63            +               db      'c'
  0083'   63            +               db      'c'
  0084'   63            +               db      'c'
  0085'   63            +               db      'c'
```

Six bytes of 63h, the letter `c`, where the example has the six letters of `TATARA`. A mistake like this one does not stop TATARA; the listing is where to see it.

CHAPTER 31

Long symbol names and case-sensitive names

31.1 The example 239
31.2 The source 239
31.3 Building it 241
31.4 Reading the two files 241
31.5 Long symbol names across modules 243
31.6 Macro names and /C 244

The name of a symbol in TATARA can be as long as it needs to be, and every character of it counts; with `/C`, capitals and small letters count too (chapter 7). This chapter builds `NAMES`, the example in `A:\TATARA\EXAMPLES\NAMES`, which shows both in one source. It then takes long symbol names and `/C` beyond one source, to a program of several modules, and `/C` to the names of macros.

31.1 The example

`NAMES` makes no program. Its `BUILD.BAT` assembles one source twice, without `/C` and with it, and keeps the symbol table of each run in a text file. The two files are the result: they show which names TATARA took as one, and which as two.

31.2 The source

```
; NAMES.AS - names as long as you like, in either case.
;
; A SYMBOL MAY BE 255 CHARACTERS AND ALL OF THEM COUNT. M80 keeps the
; first six, which is why so much CP/M-era assembly is written in
; abbreviations. The two names below differ only in their last five
; characters and are two different symbols here. 255 is the limit, and
; the test suite has a 240-character name in it; these are 48, which is
; as long as a line can carry and still be read on an MSX screen.

first_pass_symbol_table_high_water_mark_in_bytes        defl    1024
first_pass_symbol_table_high_water_mark_in_words        defl    512

; THE SAME SIX LETTERS IN TWO CASES. Without /C they are ONE symbol,
; and the second line changes its value. With /C they are TWO, with
; values 1 and 2. The symbol table of each run says which - and DEFL
; is used rather than EQU because a redefinition is legal, so the
; source assembles cleanly in both modes.

Counter         defl    1
counter         defl    2

; DIRECTIVES, MNEMONICS AND REGISTER NAMES ARE CASE-INSENSITIVE IN
; BOTH MODES. /C is about the names you choose, not the ones the Z80
; already has.

                cseg

start:          LD      A,1
                ld      b,a
                DB      Counter
                db      counter
                ret

                end     start
```

The source has three parts:

- **Two long symbol names**, of 48 characters each, that differ only in their last five characters, `bytes` and `words`. They are two different names, with their own values, 1024 and 512.
- `Counter` and `counter`, the same seven letters in two cases, each given a value with `DEFL`. `DEFL` allows a name to be given a new value (section 12.4), so the source assembles whether the two are one name or two.
- **A few instructions**, some in capitals and some not, that use both. Instructions, registers and directives can be written in either case, with or without `/C`.

31.3 Building it

```
rem BUILD.BAT - EXAMPLE: NAMES
rem
rem One source, assembled twice, and the two symbol tables are the
rem answer.
rem
rem   /S   print the symbol table when the assembly ends
rem   /P   assemble to the screen, producing no object file
rem   /C   make symbol and macro names case-SENSITIVE
rem
rem Without /C, Counter and counter are one symbol whose value ends up
rem 2, and the table has one entry. With /C they are two symbols, 1
rem and 2, and the table has both. The default is M80s behaviour.
rem
rem The two long names are 48 characters each and differ in the last
rem five. M80 would have made them one symbol.

echo === Case-insensitive, which is the default
tatara /q /p /s names.as > insens.txt

echo === Case-sensitive
tatara /q /p /c /s names.as > sens.txt

echo === Done. Compare INSENS.TXT with SENS.TXT.
```

`/P` prints the listing on the screen, and `/S` the symbol table after it (section 5.5, chapter 19). No object file is named on the command line, so none is written. Each `>` sends what TATARA prints to a file. The example needs no include file, so the batch file does not set `TATARA`.

MSX-DOS

```
A:\TATARA\EXAMPLES\NAMES>build
=== Case-insensitive, which is the default
=== Case-sensitive
=== Done. Compare INSENS.TXT with SENS.TXT.
```

31.4 Reading the two files

Each file starts with the listing: a page heading and the source as it was written. Without `/L`, the listing has no addresses and no bytes. The symbol table follows it. This is the whole table in `INSENS.TXT`:

MSX-DOS

```
ASEG - absolute

0002h  var        Counter
0200h  var        first_pass_symbol_table_high_water_mark_in_words
0400h  var        first_pass_symbol_table_high_water_mark_in_bytes

CSEG - default code segment, 0006h bytes

0000h             start

DSEG - default data segment, 0000h bytes

(no symbols)
```

and the one in `SENS.TXT`:

MSX-DOS

```
ASEG - absolute

0001h  var        Counter
0002h  var        counter
0200h  var        first_pass_symbol_table_high_water_mark_in_words
0400h  var        first_pass_symbol_table_high_water_mark_in_bytes

CSEG - default code segment, 0006h bytes

0000h             start

DSEG - default data segment, 0000h bytes

(no symbols)
```

The two long symbol names are two symbols in both files: every character of a name counts, with or without `/C`. The files differ only in `Counter`, as figure 31.1 shows:

- Without `/C`, `Counter` and `counter` are one name. `counter defl 2` gives it a new value, 2, and the table has one line for it, spelled as it was first written.
- With `/C`, they are two names, with the values 1 and 2.

The code is the same six bytes in both runs. Without `/C`, `db Counter` and `db counter` are both 2; with it, 1 and 2.

INSENS.TXT: without /C

```
0002h  var  Counter
0200h  var  first_pass_symbol_table_high_water_mark_in_words
0400h  var  first_pass_symbol_table_high_water_mark_in_bytes
```

one symbol:
the second `DEFL`
changed its value

SENS.TXT: with /C

```
0001h  var  Counter
0002h  var  counter
0200h  var  first_pass_symbol_table_high_water_mark_in_words
0400h  var  first_pass_symbol_table_high_water_mark_in_bytes
```

two symbols,
one for each case

Figure 31.1: The absolute section of the two symbol tables.

31.5 Long symbol names across modules

A name keeps its full length in the object file (section 7.2), and TANREN matches the whole of it. `LONGUSE.AS` uses the two long symbol names of `NAMES.AS` as external names:

```
; LONGUSE.AS - uses two long names from another module.
                extrn   first_pass_symbol_table_high_water_mark_in_bytes
                extrn   first_pass_symbol_table_high_water_mark_in_words
                cseg
                ld      hl,first_pass_symbol_table_high_water_mark_in_bytes
                ld      de,first_pass_symbol_table_high_water_mark_in_words
                ret
                end
```

`LONGDEF1.AS` defines only the first of them:

```
; LONGDEF1.AS - defines only the first of the two.
                public  first_pass_symbol_table_high_water_mark_in_bytes
                cseg
first_pass_symbol_table_high_water_mark_in_bytes:
                nop
                end
```

The two names share their first 43 characters, but TANREN does not take one for the other. The one that is missing is named in full:

```
A:\TATARA\EXAMPLES\NAMES>tatara /q longuse.as longuse.tro
A:\TATARA\EXAMPLES\NAMES>tatara /q longdef1.as longdef1.tro
A:\TATARA\EXAMPLES\NAMES>tanren longuse longdef1 /o:l1.com
...
Never defined:
  first_pass_symbol_table_high_water_mark_in_words
ERROR: the symbols above were never defined.
```

`LONGDEF2.AS` defines both, each as a label with a `nop`:

```
; LONGDEF2.AS - defines both.
                public  first_pass_symbol_table_high_water_mark_in_bytes
                public  first_pass_symbol_table_high_water_mark_in_words
                cseg
first_pass_symbol_table_high_water_mark_in_bytes:
                nop
first_pass_symbol_table_high_water_mark_in_words:
                nop
                end
```

MSX-DOS

```
A:\TATARA\EXAMPLES\NAMES>tatara /q longdef2.as longdef2.tro
A:\TATARA\EXAMPLES\NAMES>tanren /m longuse longdef2 /o:l2.com
...
Symbols:
  DR 0108 first_pass_symbol_table_high_water_mark_in_words
  DR 0107 first_pass_symbol_table_high_water_mark_in_bytes
2 modules, 15 records, ends at EOF.
Wrote L2.COM, 0100-0108 (9 bytes).
```

Both names are matched, each at its own address. A program can therefore name its routines and variables for what they are, as `print_string` or `keyboard_buffer_length`, across all its modules. The only limits are 255 characters and the length of a line (section 7.2).

31.6 Macro names and /C

`/C` applies to the names of macros as well as to symbols (section 16.8). `MACCASE.AS` defines a macro called `Twice`, and uses it once in each case:

```
; MACCASE.AS - a macro defined as Twice and used as twice.
Twice           macro
                nop
                nop
                endm
                cseg
                Twice
                twice
                end
```

Without `/C`, both lines are the macro, and each becomes two `nop`s:

MSX-DOS

```
A:\TATARA\EXAMPLES\NAMES>tatara /p /s maccase.as
...
                cseg
                Twice
                nop
                nop
                twice
                nop
                nop
                end
...
CSEG - default code segment, 0004h bytes
```

With `/C`, `twice` is not the name of the macro, nor of any instruction or directive:

MSX-DOS

```
A:\TATARA\EXAMPLES\NAMES>tatara /p /c /s maccase.as
...
MACCASE.AS(8): ERROR: not a directive, a macro or an instruction.
```

A source written for `/C` has to use each macro in exactly the case of its definition, including the macros of the include files, such as `system`.

CHAPTER 32

A binary for MSX-BASIC's BLOAD

32.1 The example 247

32.2 The source 247

32.3 Building it 249

32.4 Running it from BASIC 250

32.5 Where the file can go. 250

32.6 Things to try. 252

MSX-BASIC can load a file of machine code with `BLOAD` and run it. This chapter builds `BINARY`, the example in `A:\TATARA\EXAMPLES\BINARY`, which makes such a file, runs it from BASIC, and then looks at where in BASIC's memory a machine-code file can safely go.

32.1 The example

`BINARY` is one source, `BEEP.AS`, and its `BUILD.BAT`. The program prints a line, sounds the MSX's buzzer and returns to BASIC. It is not a `.COM` program: MSX-DOS is not there when BASIC runs it, so it prints through the MSX's BIOS, the routines in the MSX's ROM. TANREN writes it with a `BLOAD` header (section 26.4), which makes it a `.BIN` file.

32.2 The source

```
; BEEP.AS - a machine-code file for MSX BASIC.
;
; BLOAD"BEEP.BIN",R loads this and runs it. The seven bytes BASIC
; reads first - FEh, the first address, the LAST address, and the
; address to call - are written by TANREN when /B is given, and
; nothing else makes a file a .BIN.
;
; ASEG and ORG: the code is at a fixed address and the linker moves
; nothing. It has to be, because BASIC loads it where the header says
; and does not relocate anything.
;
; There is no MSX-DOS here, so the printing is the BIOS.

                include bios.inc        ; CHPUT, BEEP, and every other
                                        ;   MAIN ROM entry
                include ascii.inc       ; CHR_CR and CHR_LF

                aseg
                org     8000h           ; above BASIC in a 64 KB machine

start:          ld      hl,msg
loop:           ld      a,(hl)
                or      a
                jr      z,done
                push    hl
                call    CHPUT
                pop     hl
                inc     hl
                jr      loop
done:           call    BEEP
                ret                     ; back to BASIC, which is waiting

msg:            db      "Loaded by BLOAD, run from BASIC.",CHR_CR,CHR_LF,0

                end     start           ; the address the header carries
```

- **`ASEG` and `ORG`** put the code in the absolute segment, at 8000h (chapter 10). The file is loaded at the address in its header, and TANREN gives it the address the source names.
- **The loop** prints the message one character at a time with `CHPUT`, the BIOS routine that prints the character in `A`, until it reaches the zero byte at the end. `BIOS.INC` gives the addresses of `CHPUT` and `BEEP`.
- `BEEP`, another BIOS routine, sounds the buzzer, and `RET` returns to BASIC.
- **The `END` line** names `start`. TANREN puts it in the header as the address to call (section 26.5).

32.3 Building it

```
rem BUILD.BAT - EXAMPLE: BINARY
rem
rem A .BIN for MSX BASIC instead of a .COM for MSX-DOS. Two things
rem make the difference:
rem
rem   ASEG and ORG   the code is at a fixed address, not relocatable
rem   TANREN /B      the seven-byte BLOAD header goes in front of it
rem
rem The address BASIC calls is the one on the END line of the source.
rem Without /B the same link writes the bytes and no header, which is
rem what a ROM needs - see the ROM example.
rem
rem To run it, from BASIC:  BLOAD then the file name in quotes, ,R

rem The headers are in A:\TATARA\INCLUDE and are not copied here.
rem TATARA is rule 3 of the include search - see the DIRS example.
rem CHANGE THE PATH BELOW if you put the tree somewhere else.

set TATARA=a:\tatara\include

echo === Assembling
tatara /q beep.as beep.tro

echo === Linking, with a BLOAD header
tanren /q /b /o:beep.bin beep.tro

echo === Done. BEEP.BIN is 7 bytes of header and the rest is code.

set TATARA=
```

MSX-DOS

```
A:\TATARA\EXAMPLES\BINARY>build
=== Assembling
=== Linking, with a BLOAD header
=== Done. BEEP.BIN is 7 bytes of header and the rest is code.
```

Without `/Q`, TANREN's summary shows what it wrote:

MSX-DOS

```
A:\TATARA\EXAMPLES\BINARY>tanren /b /o:beep.bin beep.tro
...
1 modules, 6 records, ends at EOF.
Wrote BEEP.BIN, 8000-8035 (54 bytes), BLOAD header, entry 8000.
```

The program is 54 bytes, and the file 61. Its first seven bytes are the header:

```
FE  00 80  35 80  00 80
```

`FE`, then the first address, 8000h, the last, 8035h, and the address to call, 8000h, each with its low byte first. The program follows: `21 13 80`, `ld hl,msg`, with the message at 8013h.

32.4 Running it from BASIC

In MSX-BASIC, from the example's directory, `BLOAD` with `,R` loads the file and calls its start:

MSX-DOS

```
Ok
BLOAD"BEEP.BIN",R
Loaded by BLOAD, run from BASIC.
Ok
```

The buzzer sounds after the line, and `RET` returns to BASIC, which prints `Ok`.

32.5 Where the file can go

While BASIC runs, the addresses below 8000h hold its ROM and the BIOS, so a file for `BLOAD` has to go in the RAM above them. But BASIC uses that RAM too: the BASIC program in memory starts just above 8000h, and its variables follow it. The example worked above because no program was in memory. With one, the same file overwrites it:

MSX-DOS

```
list
10 PRINT "hello"
Ok
bload"beep.bin",r
Loaded by BLOAD, run from BASIC.
Ok
list
46974 INSTRMAXTRON
25697 ed by BLOAD, run from BASIC.501WAIT_VPOKE=^KILLL<0MERGEeMAXO<>41933;+;(3:(
...
```

The machine code still ran and printed its line, but the BASIC program is gone: `LIST` shows BASIC reading the machine code as if it were a program, and then the machine stops responding.

`CLEAR` keeps BASIC away from the memory a file needs. `CLEAR 200,&HC000` gives BASIC 200 bytes for its strings and, more to the point here, tells it to use no memory from C000h upwards. A file at C000h is then out of BASIC's way, as figure 32.1 shows.

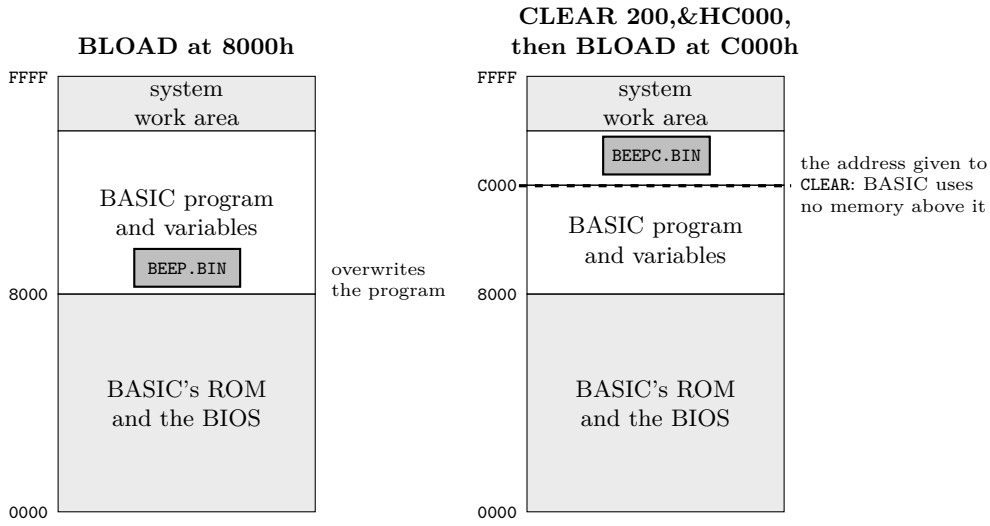


Figure 32.1: MSX-BASIC's memory, with the file at 8000h and at C000h.

`BEEPC.AS` is `BEEP.AS` with its `ORG` changed:

```
                aseg
                org     0C000h          ; above the CLEAR address
```

MSX-DOS

```
Ok
clear 200,&hc000
Ok
bload"beepc.bin",r
Loaded by BLOAD, run from BASIC.
Ok
```

Give the `CLEAR` before the first `BLOAD`, and before the BASIC program is typed or loaded: `CLEAR` also clears the variables. A BASIC program that uses machine code usually starts with the `CLEAR` line and then `BLOAD`s its files. The more machine code there is, the lower the address given to `CLEAR` has to be, and the less memory is left for BASIC.

32.6 Things to try

The files below are not part of the example: make them in its directory, with `TATARA` set as in `BUILD.BAT`.

32.6.1 Let TANREN place the code

A module does not have to use `ASEG` and `ORG`. `BEEPR.AS` is the same program in the default code segment, and TANREN places it with `/P:` (section 24.3):

```
; BEEPR.AS - the same code, relocatable: TANREN /P: places it.
                include bios.inc
                include ascii.inc
                cseg
start:          ld      hl,msg
loop:           ld      a,(hl)
                or      a
                jr      z,done
                push    hl
                call    CHPUT
                pop     hl
                inc     hl
                jr      loop
done:           call    BEEP
                ret
msg:            db      "Relocated by TANREN /P:.",CHR_CR,CHR_LF,0
                end     start
```

MSX-DOS

```
A:\TATARA\EXAMPLES\BINARY>tatara /q beepr.as beepr.tro
A:\TATARA\EXAMPLES\BINARY>tanren /b /p:c000 /o:beepr.bin beepr.tro
...
Wrote BEEPR.BIN, C000-C02D (46 bytes), BLOAD header, entry C000.
```

The same source can then be linked for any address, and a program of several modules can be placed as a whole.

32.6.2 Call it with USR

Without `,R`, `BLOAD` only loads the file. `DEFUSR` names the address of a machine-code routine, and `USR` calls it:

MSX-DOS

```
Ok
clear 200,&hc000
Ok
bload"beepr.bin"
Ok
defusr=&hc000
Ok
a=usr(0)
Relocated by TANREN /P:.
Ok
```

The routine can then be called as often as the program needs, without loading it again.

32.6.3 Leave out /B

Linked without `/B`, the file has the same 54 bytes and no header:

MSX-DOS

```
A:\TATARA\EXAMPLES\BINARY>tanren /o:beepraw.bin beep.tro
...
Wrote BEEPRAW.BIN, 8000-8035 (54 bytes), entry 8000.
```

`BLOAD` refuses it, because it does not start with `FE`:

MSX-DOS

```
Ok
bload"beepraw.bin",r
Bad file mode
Ok
```

CHAPTER 33

A 16 KB ROM cartridge

33.1 The example 255
33.2 The source 255
33.3 Building it 258
33.4 Running it 259
33.5 Things to try 259

A cartridge is a program in ROM that the MSX finds and starts on its own, when it is switched on. This chapter builds `ROM`, the example in `A:\TATARA\EXAMPLES\ROM`, which makes the image of a 16 KB cartridge: the header the MSX looks for, a routine that prints a line, and the padding that makes the file the right size. It then runs the image in an emulator, and changes it in three ways.

33.1 The example

A cartridge starts before MSX-DOS or MSX-BASIC does: the MSX looks at each slot, finds the cartridge, and calls it while it is still starting. The program can use the BIOS, which is always there, but nothing that MSX-DOS or BASIC provides. `ROM` sets up a text screen, prints a line, and stays there.

The example has two files, `ROM.AS` and `BUILD.BAT`. The result is a file, `ROM.ROM`, that holds the 16 KB of the cartridge, byte for byte. It is not a program that MSX-DOS can run: it goes into an emulator as a cartridge, or onto a flash cartridge.

33.2 The source

```
; ROM.AS - a 16 KB cartridge.
;
; A cartridge in page 1 begins at 4000h with a sixteen-byte header.
; The BIOS looks for the letters AB, and calls the INIT address if it
; is not zero - before there is an operating system of any kind.
;
; NOTHING HERE MAY CALL MSX-DOS. There is no MSX-DOS yet. The screen
; and the printing are the BIOS, and the code never returns, because
; there is nothing to return to.
;
; THE LAST TWO LINES ARE WHAT MAKES THE FILE 16 KB. TANREN writes the
; span from the lowest byte of content to the highest, and a gap
; inside that span comes out as zeros. One byte at 7FFFh therefore
; makes the file exactly 16384 bytes, which is what a cartridge has
; to be. A DS would not do it: the space a DS reserves after the last
; byte of content is deliberately not in the file.

                include bios.inc        ; INITXT, CHPUT, and the rest
                include ascii.inc       ; CHR_CR and CHR_LF

                aseg
                org     4000h

                db      "AB"            ; a cartridge, says the BIOS
                dw      init            ; INIT: called at once
                dw      0               ; STATEMENT: no CALL statements
                dw      0               ; DEVICE: no device
                dw      0               ; TEXT: no BASIC program in here
                dw      0,0,0           ; reserved, and zero

init:           call    INITXT          ; a text screen, from nothing
                ld      hl,msg
ini.lp:         ld      a,(hl)
                or      a
                jr      z,ini.end
                push    hl
                call    CHPUT
                pop     hl
                inc     hl
                jr      ini.lp
ini.end:        jr      ini.end         ; a cartridge has nowhere to go

msg:            db      "Assembled by Tatara. Linked by TANREN.",CHR_CR,CHR_LF,0

                org     7fffh           ; the last byte of the 16 KB...
                db      0               ;   ...so that the file is all of it

                end
```

33.2.1 The header

A 16 KB cartridge sits at 4000h to 7FFFh, and its first sixteen bytes are a header that tells the MSX what the cartridge holds (figure 33.1):

- **ID**, two bytes: the letters `AB`. Without them, the MSX does not take the ROM for a cartridge.

- **INIT**, the address of a routine the MSX calls when it starts, or 0000h for none. Here it is `init`.
- **STATEMENT**, **DEVICE** and **TEXT**, the addresses of routines that add `CALL` statements and devices to BASIC, and of a BASIC program in the ROM. This cartridge has none of them, and they are 0000h.
- **Six reserved bytes**, which must be 00h.

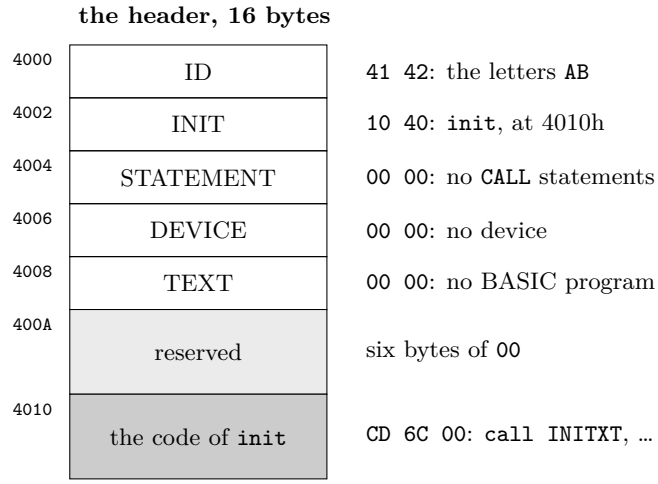


Figure 33.1: The header of `ROM.ROM`, at 4000h, and the value of each field.

The MSX looks for the header at 4000h and at 8000h (section 26.6). `ASEG` and `ORG 4000h` put it at 4000h, with nothing for TANREN to move.

33.2.2 INIT

`init` calls `INITXT`, the BIOS routine that sets up the text screen, prints the message with `CHPUT`, and then jumps to itself for ever: `ini.end: jr ini.end`. The MSX stays on the screen the cartridge left it on. This is how a game, for example, takes the machine over.

33.2.3 The last two lines

A cartridge image has to hold all its 16 KB. TANREN writes the memory from the first byte with content to the last (section 24.1.1), so the program alone would make a file of 77 bytes. `org 7fffh` and `db 0` put a byte at the last address, and everything between the message and 7FFFh is written as zeros.

33.3 Building it

```
rem BUILD.BAT - EXAMPLE: ROM
rem
rem A 16 KB cartridge image. No MSX-DOS anywhere in it: the code runs
rem from a cartridge slot before there is an operating system, so it
rem calls the BIOS and never returns.
rem
rem   ASEG and ORG 4000h   page 1, where a 16 KB cartridge lives
rem   no /B                a ROM has no header of any kind
rem   ORG 7FFFH and DB 0   what makes the file exactly 16384 bytes
rem
rem THIS IS THE ONE EXAMPLE A SUCCESSFUL BUILD DOES NOT PROVE. Load
rem ROM.ROM as a cartridge in an emulator, or write it to a flash
rem cartridge. It is not a program whose name you can type.

rem The headers are in A:\TATARA\INCLUDE and are not copied here.
rem TATARA is rule 3 of the include search - see the DIRS example.
rem CHANGE THE PATH BELOW if you put the tree somewhere else.

set TATARA=a:\tatara\include

echo === Assembling
tatara /q rom.as rom.tro

echo === Linking, with no header of any kind
tanren /q /o:rom.rom rom.tro

echo === Done. ROM.ROM should be exactly 16384 bytes.

set TATARA=
```

The link has no `/B`: the file is the cartridge's own bytes, with no `BLOAD` header in front of them. Its only header is the one in the source.

MSX-DOS

```
A:\TATARA\EXAMPLES\ROM>build
=== Assembling
=== Linking, with no header of any kind
=== Done. ROM.ROM should be exactly 16384 bytes.
```

Without `/Q`, TANREN's summary confirms the size:

MSX-DOS

```
A:\TATARA\EXAMPLES\ROM>tanren /o:rom.rom rom.tro
...
1 modules, 6 records, ends at EOF.
Wrote ROM.ROM, 4000-7FFF (16384 bytes).
```

The file starts with `41 42 10 40`, `AB` and the address of `init`, 4010h, low byte first, followed by the ten zero bytes of the other fields. `init` begins at 4010h with `CD 6C 00`, `call INITXT`.

33.4 Running it

Insert `ROM.ROM` as a cartridge in an emulator and reset the MSX. In openMSX, for example, the console command `carta` inserts a ROM image in the first cartridge slot, and `reset` restarts the machine. The screen shows the message on a text screen, and nothing else:

MSX-DOS

```
Assembled by Tatara. Linked by TANREN.
```

The MSX found `AB` at 4000h, called `init`, and never came back from it. Neither MSX-DOS nor BASIC starts.

33.5 Things to try

The files below are not part of the example: make them in its directory, with `TATARA` set as in `BUILD.BAT`, and link each one as `ROM.AS` is linked.

33.5.1 Return from INIT

`ROMRET.AS` is `ROM.AS` with `ret` in place of the endless jump:

```
ini.end:        ret                     ; and the MSX goes on starting
```

The message appears for a moment, and then the MSX goes on starting as it would without the cartridge; on an MSX with a disk drive, MSX-DOS starts. The comment at the top of `ROM.AS` says there is nothing to return to, but the MSX does wait for `INIT` to return. A cartridge that only adds something to the system, and leaves the rest of the start to it, sets up what it needs and returns.

33.5.2 Spoil the ID

`ROMBAD.AS` is `ROM.AS` with `"AC"` in place of `"AB"`:

```
                db      "AC"            ; not a cartridge ID
```

It links into a 16 KB file like the others, but the MSX does not take it for a cartridge. Nothing of it is seen: the MSX starts MSX-DOS as if the slot were empty. When a cartridge does nothing at all, check the first two bytes of the file.

33.5.3 Let TANREN place the cartridge

A cartridge does not have to use `ASEG`. `ROMREL.AS` is the same cartridge in the default code segment, placed at 4000h by `/P:` (section 24.3):

```
; ROMREL.AS - the cartridge as a relocatable module, placed by /P:4000.
                include bios.inc
                include ascii.inc
                cseg
header:         db      "AB"
                dw      init
                dw      0,0,0
                dw      0,0,0
init:           call    INITXT
                ld      hl,msg
ini.lp:         ld      a,(hl)
                or      a
                jr      z,ini.end
                push    hl
                call    CHPUT
                pop     hl
                inc     hl
                jr      ini.lp
ini.end:        jr      ini.end
msg:            db      "Placed by TANREN /P:4000.",CHR_CR,CHR_LF,0
                ds      3FFFh-($-header)
                db      0
                end
```

`ORG` cannot pad a relocatable segment to a fixed address, so the padding is worked out from the size: `$-header` is the number of bytes so far, and the `DS` reserves the rest of the 16 KB but one. The `DB` after it fills the last byte, so that the `DS` is inside the file and not after its end.

MSX-DOS

```
A:\TATARA\EXAMPLES\ROM>tatara /q romrel.as romrel.tro
A:\TATARA\EXAMPLES\ROM>tanren /p:4000 /o:romrel.rom romrel.tro
...
1 modules, 7 records, ends at EOF.
Wrote ROMREL.ROM, 4000-7FFF (16384 bytes).
```

As a cartridge, it prints its own line and stays there:

MSX-DOS

```
Placed by TANREN /P:4000.
```

Written this way, the code of a cartridge can be spread over several modules, as any other program, and TANREN places them all.

CHAPTER 34

A project in several directories

34.1 The example 263
34.2 The sources 264
34.3 Building and running it 266
34.4 Things to try 267

A program of any size soon outgrows one directory: its sources in one place, the files it includes beside them, headers shared with other programs somewhere else, and perhaps the object files apart from all of them. This chapter builds `DIRS`, the example in `A:\TATARA\EXAMPLES\DIRS`, whose files are spread over three directories, and follows how TATARA finds each of them (chapter 13). It then builds the same program in other ways: from another directory, with a relative directory in `TATARA`, and with the object file and a link file in a directory of their own.

34.1 The example

`DIRS` prints three lines. Each one is in a different include file, and says which rule of the search found that file:

MSX-DOS

```
A:\TATARA\EXAMPLES\DIRS>dirs
Rule 1: beside the file that asked.
Rule 1 again, one level deeper.
Rule 3: found through TATARA.
```

TATARA looks for an include file in three places, in this order (section 13.2):

1. the directory of the file that holds the `INCLUDE`;
2. the current directory;

3. each directory named in TATARA.

Figure 34.1 shows the example's files, and the rule that finds each one when the program is built from DIRS.

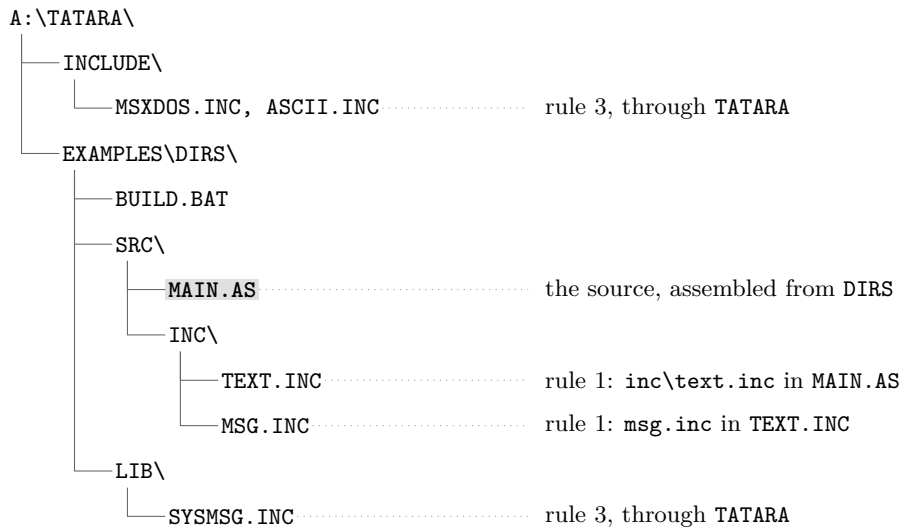


Figure 34.1: The files of DIRS, and the rule that finds each include file.

34.2 The sources

SRC\MAIN.AS is the module:

```
; SRC\MAIN.AS - a program whose parts are in three directories.
;
; AN INCLUDE IS LOOKED FOR IN THREE PLACES, IN THIS ORDER:
;
;   1. the directory of the file doing the including
;   2. the current directory
;   3. each directory named in the TATARA environment variable
;
; Rule 1 comes first so that a file always finds the header sitting
; beside it, whatever directory you happen to be standing in when you
; run the build. Neither line below depends on where that is.

                include ascii.inc       ; CHR_CR and CHR_LF, for the messages
                include msxdos.inc      ; rule 3: TATARA names the
                                        ;   directory it is in

                cseg

start:          ld      de,text
                call    prtstr
```

```
                ld      de,deeper
                call    prtstr
                ld      de,sysmsg
                call    prtstr
                system  _TERM0

prtstr:         ld      c,_STROUT
                jp      BDOS            ; and BDOS returns to the caller

; THE MESSAGES COME AFTER THE CODE. Each include below holds a DB,
; and MSX-DOS starts a .COM at 0100h: text there would be run as
; instructions.

                include inc\text.inc    ; rule 1, with a separator:
                                        ;   relative to THIS file
                include sysmsg.inc      ; rule 3 as well, from the OTHER
                                        ;   directory in TATARA

                end     start
```

It includes four files, in two groups:

- `ASCII.INC` and `MSXDOS.INC`, at the top, define names and the `system` macro and produce no bytes. Neither is beside `MAIN.AS` or in the current directory, so rule 3 finds them in `A:\TATARA\INCLUDE`, the first directory in `TATARA`.
- `INC\TEXT.INC` and `SYSMSG.INC`, after the code, each hold a message. They come after the code because they produce bytes: placed before `start`, the text would be at 0100h, and MSX-DOS would run it as instructions when the program starts (section 21.1).

`inc\text.inc` has a directory in its name, but no drive and no leading `\`, so it is relative (section 13.4), and rule 1 looks for it relative to `MAIN.AS`: in `SRC\INC`. `SRC\INC\TEXT.INC` holds the first message, and includes `MSG.INC`:

```
; SRC\INC\TEXT.INC - reached as inc\text.inc from SRC\MAIN.AS, so
; rule 1 found it: a name with a separator in it is still relative to
; the file that asked for it.
;
; The include below has no directory at all, and rule 1 finds it
; beside THIS file rather than beside MAIN.AS.

                include msg.inc

text:           db      "Rule 1: beside the file that asked.",CHR_CR,CHR_LF,"$"
```

Rule 1 for this `INCLUDE` is the directory of `TEXT.INC`, so `MSG.INC` is found in `SRC\INC` too:

```
; SRC\INC\MSG.INC - a bare name, included from TEXT.INC, which lives
; in this directory.
;
; IF THE SEARCH STARTED IN THE CURRENT DIRECTORY, THIS WOULD NOT BE
; FOUND: the build runs two levels up, and there is no MSG.INC there.

deeper:         db      "Rule 1 again, one level deeper.",CHR_CR,CHR_LF,"$"
```

`LIB\SYSMSG.INC` is in neither of those directories, nor in the current one. It is found because `BUILD.BAT` puts `LIB` in `TATARA`:

```
; LIB\SYSMSG.INC - not beside MAIN.AS, and not in the directory the
; build runs from. BUILD.BAT puts this directory in TATARA, which is
; rule 3, and that is how a project keeps its shared headers in one
; place instead of a copy per program.

sysmsg:         db      "Rule 3: found through TATARA.",CHR_CR,CHR_LF,"$"
```

34.3 Building and running it

```
rem BUILD.BAT - EXAMPLE: DIRS
rem
rem Sources in SRC, headers beside them in SRC\INC, shared headers in
rem LIB. The build runs from HERE and nothing is copied anywhere.
rem
rem TATARA holds the directories rule 3 looks in - the same idea as
rem PATH, and the same shape: several may be separated by semicolons.
rem It is set for the build and cleared afterwards so that nothing
rem outside this example inherits it.
rem
rem THE PATH BELOW IS ABSOLUTE and names this example. MSX-DOS gives a
rem batch file no way to ask where it is, so change this line if you
rem put the tree somewhere else.

set TATARA=a:\tatara\include;a:\tatara\examples\dirs\lib

echo === Assembling SRC\MAIN.AS
tatara /q src\main.as dirs.tro

echo === Linking
tanren /q /o:dirs.com dirs.tro

set TATARA=

echo === Done. Type DIRS to run it.
```

`TATARA` names two directories, separated by a semicolon (section 13.3). The batch file is run from `DIRS`, and names the source as `src\main.as`; the object file and the program are written in `DIRS`.

MSX-DOS

```
A:\TATARA\EXAMPLES\DIRS>build
=== Assembling SRC\MAIN.AS
=== Linking
=== Done. Type DIRS to run it.
A:\TATARA\EXAMPLES\DIRS>dirs
Rule 1: beside the file that asked.
Rule 1 again, one level deeper.
Rule 3: found through TATARA.
```

Without `/Q`, TANREN shows that the program starts at 0100h, with the code, as it must:

MSX-DOS

```
A:\TATARA\EXAMPLES\DIRS>tanren /o:dirs.com dirs.tro
...
1 modules, 7 records, ends at EOF.
Wrote DIRS.COM, 0100-0183 (132 bytes), entry 0100.
```

34.4 Things to try

Each of these starts from `DIRS`, with `TATARA` set as in `BUILD.BAT` unless it says otherwise.

34.4.1 Leave LIB out of TATARA

With only the include directory in `TATARA`, nothing finds `SYSMSG.INC`:

MSX-DOS

```
A:\TATARA\EXAMPLES\DIRS>set tatara=a:\tatara\include
A:\TATARA\EXAMPLES\DIRS>tatara src\main.as nolib.tro
...
ERROR: cannot open SYSMSG.INC
    included from SRC\MAIN.AS(36)
```

The message names the file as it was written, and the line of `MAIN.AS` that asked for it.

34.4.2 A relative directory in TATARA

A directory in `TATARA` can be relative, like any other path:

MSX-DOS

```
A:\TATARA\EXAMPLES\DIRS>set tatara=a:\tatara\include;lib
A:\TATARA\EXAMPLES\DIRS>tatara src\main.as rel.tro
...
ended at SRC\MAIN.AS(39)
```

`lib` is taken from the current directory, so this works only while the build runs from `DIRS`. `BUILD.BAT` uses an absolute path, so that `TATARA` means the same wherever the build runs.

34.4.3 Build from another directory

Rule 1 does not depend on the current directory. Assembled from inside `SRC`, `MAIN.AS` still finds `INC\TEXT.INC` and `MSG.INC`, and `TATARA` gives the rest:

MSX-DOS

```
A:\TATARA\EXAMPLES\DIRS>cd src
A:\TATARA\EXAMPLES\DIRS\SRC>tatara main.as ..\dirs2.tro
...
ended at MAIN.AS(39)
A:\TATARA\EXAMPLES\DIRS\SRC>cd ..
A:\TATARA\EXAMPLES\DIRS>tanren /o:dirs2.com dirs2.tro
...
Wrote DIRS2.COM, 0100-0183 (132 bytes), entry 0100.
A:\TATARA\EXAMPLES\DIRS>dirs2
Rule 1: beside the file that asked.
Rule 1 again, one level deeper.
Rule 3: found through TATARA.
```

34.4.4 Keep the object files apart

A larger project often keeps its object files, and the link file that names them, in a directory of their own. `OBJ\DIRS.LNK` is such a link file:

```
; DIRS.LNK - the link, kept with the object files.
main
/o:dirs3.com
```

TATARA writes the object file into `OBJ`, and TANREN, given the link file, looks for `MAIN.TRO` beside it first (section 23.1):

MSX-DOS

```
A:\TATARA\EXAMPLES\DIRS>tatara /q src\main.as obj\main.tro
A:\TATARA\EXAMPLES\DIRS>tanren @obj\dirs
...
1 modules, 7 records, ends at EOF.
Wrote DIRS3.COM, 0100-0183 (132 bytes), entry 0100.
A:\TATARA\EXAMPLES\DIRS>dirs3
Rule 1: beside the file that asked.
Rule 1 again, one level deeper.
Rule 3: found through TATARA.
```

`DIRS3.COM` is written in the current directory, `DIRS`, not in `OBJ`: `/O:` uses the name exactly as it is written (section 21.2). To put the program somewhere else, give the path in the name, as in `/o:bin\dirs3.com`.

CHAPTER 35

Building TATARA and TANREN from their own sources

35.1 The sources 271
35.2 BUILD.BAT 272
35.3 Running it 275
35.4 Building the tools with themselves 276

TATARA and TANREN are written in the source language that TATARA assembles, and they are built on an MSX, by TATARA and TANREN: each release is made by the release before it. This chapter shows where the sources are and how they are laid out, goes through the batch file that builds the two tools, runs it, and then builds the tools a second time, with the tools that the first build made.

How the tools work inside is not covered here. The comments in the sources describe each module.

35.1 The sources

The archive that chapter 3 installs holds the two programs, the include files, the examples, `README.TXT` and `LICENSE`, but not the sources. The sources are in a public repository, at `https://github.com/javilm/tatara`, which can be downloaded as a ZIP file and copied to the MSX.

Figure 35.1 shows the parts of the tree that the build uses. This chapter puts it in `A:\SRC\TATARA`; any other place works, as long as the directories inside it stay as they are.

The repository holds more than the build needs: the include files of the distribution, the examples of this manual, and the test suites of the two tools. `BUILD\`

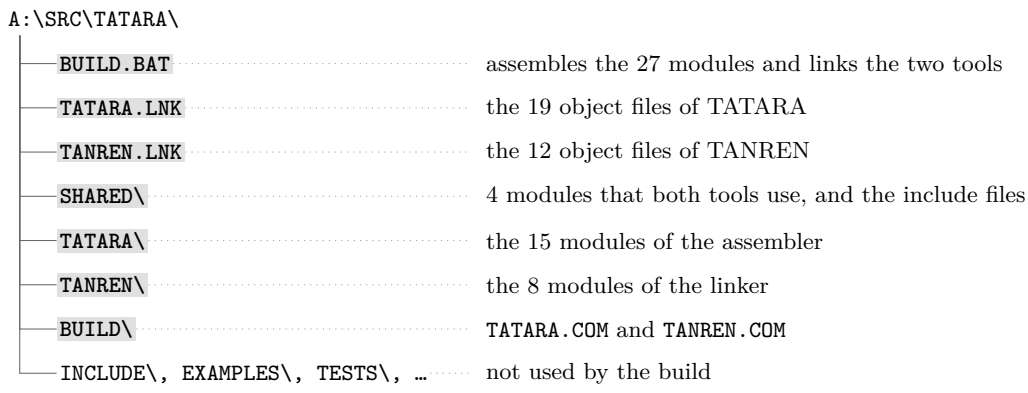


Figure 35.1: The source tree, and what the build uses from it.

holds the programs of the current release, and the build writes its own over them.

> Note. The sources take about 700 KB, and the build needs space for its object files and its two programs as well. That is more than a 720 KB floppy disk holds: build the tools on a hard disk, or on a CF or SD card.

The build is run by the tools that are already installed, so `A:\TATARA\BIN`, or wherever `TATARA.COM` and `TANREN.COM` are, must be in `PATH` (section 3.4).

35.2 BUILD.BAT

`BUILD.BAT` assembles the 27 modules and links the two tools:

```
rem BUILD.BAT - assemble and link both tools, with Tatara.

rem SILENCE IS SUCCESS. /Q prints nothing for a module that works and
rem errors print anyway, so anything on the screen between the ===
rem lines is a problem.

set TATARA=shared\


echo === Assembling TATARA modules...
tatara /q tatara\tatara.as tatara.tro
tatara /q tatara\cmdline.as cmdline.tro
tatara /q tatara\srcline.as srcline.tro
tatara /q tatara\fields.as fields.tro
tatara /q tatara\dirtab.as dirtab.tro
tatara /q tatara\macros.as macros.tro
tatara /q tatara\errs.as errs.tro
tatara /q tatara\expand.as expand.tro
```

```
tatara /q tatara\cond.as cond.tro
tatara /q tatara\expr.as expr.tro
tatara /q tatara\symtab.as symtab.tro
tatara /q tatara\optab.as optab.tro
tatara /q tatara\insn.as insn.tro
tatara /q tatara\emit.as emit.tro
tatara /q tatara\objout.as objout.tro

echo === Assembling TANREN modules...
tatara /q tanren\tanren.as tanren.tro
tatara /q tanren\lcmd.as lcmd.tro
tatara /q tanren\lobj.as lobj.tro
tatara /q tanren\lseg.as lseg.tro
tatara /q tanren\lsym.as lsym.tro
tatara /q tanren\lerrs.as lerrs.tro
tatara /q tanren\limg.as limg.tro
tatara /q tanren\arglist.as arglist.tro

echo === Assembling shared modules...
tatara /q shared\msxdos.as msxdos.tro
tatara /q shared\strutil.as strutil.tro
tatara /q shared\alloc.as alloc.tro
tatara /q shared\hash.as hash.tro

echo === Linking TATARA and TANREN binaries
tanren /o:build\tatara.com @tatara.lnk
tanren /o:build\tanren.com @tanren.lnk

echo === Cleaning up
del *.tro
```

It is run from the root of the tree, and does four things.

- **It sets `TATARA`** to `shared\`. The modules include their own `.INC` files, which are beside them, and rule 1 of the include search finds those. They also include the files in `SHARED`, such as `MSXDOS.INC`, and rule 3 finds those through `TATARA` (section 13.2). The directory is relative, so it is taken from the current directory, and the build works only from the root of the tree (chapter 34).
- **It assembles the 27 modules**, one **`tatara`** line each, in three groups: the 15 modules of the assembler, the 8 of the linker, and the 4 that both use. Each object file is written in the root of the tree.
- **It links the two tools**, each from a link file (chapter 22), with `/O:` to write the program into `BUILD`.
- **It deletes the object files.**

Every **`tatara`** line has `/Q`. A module that assembles without an error prints

nothing, and errors are printed with or without `/Q`, so the comment at the top says it: anything on the screen between the `===` lines is a problem. The two links have no `/Q`, and print TANREN's summary.

`TATARA.LNK` names the 19 object files of the assembler, and section 22.2 shows it. `TANREN.LNK` names the 12 of the linker:

```
; the linker's own modules.

tanren.tro                              ; the driver
lcmd.tro arglist.tro                    ; the command line
lobj.tro                                ; the object reader
lseg.tro lsym.tro limg.tro              ; tables and the image
lerrs.tro

msxdos.tro strutil.tro alloc.tro hash.tro       ; shared
```

Both files end with the same line: the four shared modules are assembled once, and linked into both tools. Each file names the driver of its tool first, so that its code is at 0100h, where MSX-DOS starts a `.COM` program (section 21.1). Figure 35.2 shows how the 27 modules become the two programs.

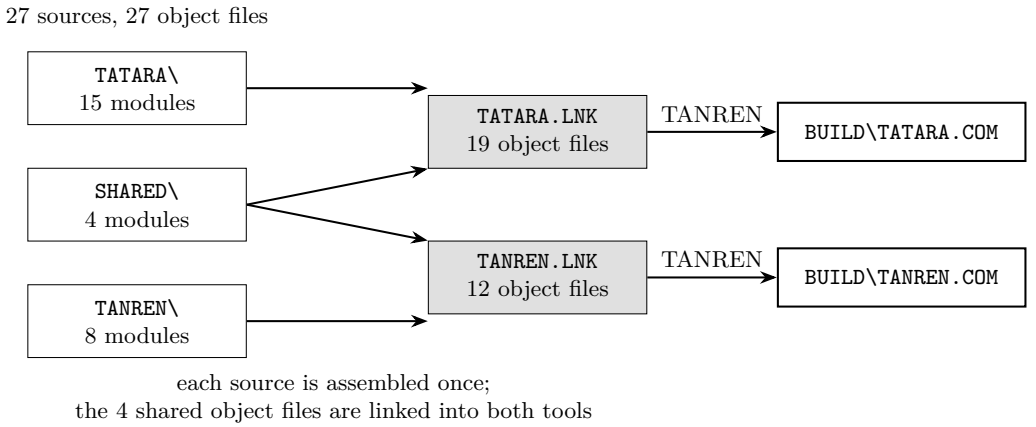


Figure 35.2: The 27 modules, the two link files and the two programs.

Note. `BUILD.BAT` does not set `TATARA` back when it ends, so after a build it holds `shared\`. Set it to your include directory again before you assemble anything else (section 3.4).

35.3 Running it

Run `BUILD.BAT` from the root of the tree:

MSX-DOS

```
A:\SRC\TATARA>build
=== Assembling TATARA modules...
=== Assembling TANREN modules...
=== Assembling shared modules...
=== Linking TATARA and TANREN binaries
Tatara MSX Linker v1.2.0
Copyright (C) 2026 Javier Lavandeira
https://tatara.tools

19 modules, 1898 records, ends at EOF.
Wrote BUILD\TATARA.COM, 0100-7990 (30865 bytes).
Tatara MSX Linker v1.2.0
Copyright (C) 2026 Javier Lavandeira
https://tatara.tools

12 modules, 766 records, ends at EOF.
Wrote BUILD\TANREN.COM, 0100-3501 (13314 bytes).
=== Cleaning up
```

Nothing is printed between the `===` lines of the assembly, so all 27 modules assembled without an error. The `Wrote` lines have no `entry` part, because no module of either tool names an address on its `END` line; each program starts at 0100h, with the code of its first module (section 21.4).

How long the build takes depends on the machine, and on the interface and the card that the files are on. As a guide, building from a CF or SD card in an IDE interface, it takes about 4 minutes on an MSX turbo R (a Panasonic FS-A1GT), and about 23 minutes on an MSX2+ (a Sony HB-F1XV). The turbo R is faster because of its R800 processor.

An emulator such as openMSX can also run the emulated MSX as fast as the computer it runs on allows, instead of at the speed of a real MSX. The build then takes much less time: about 14 seconds on a recent Mac.

`/V` checks that the new programs run:

MSX-DOS

```
A:\SRC\TATARA>build\tatara /v
Tatara MSX Macro-Assembler v1.2.0
Copyright (C) 2026 Javier Lavandeira
https://tatara.tools


A:\SRC\TATARA>build\tanren /v
Tatara MSX Linker v1.2.0
Copyright (C) 2026 Javier Lavandeira
https://tatara.tools


A:\SRC\TATARA>
```

35.4 Building the tools with themselves

The first build was made by the installed tools. Building again with the tools it made shows that they can build themselves. Keep a copy of them in a directory of their own, `STAGE1`, and put that directory first in `PATH`, so that MSX-DOS finds the new `TATARA.COM` and `TANREN.COM` before the installed ones:

MSX-DOS

```
A:\SRC\TATARA>md stage1
A:\SRC\TATARA>copy build\tatara.com stage1
A:\SRC\TATARA>copy build\tanren.com stage1
A:\SRC\TATARA>set path=a:\src\tatara\stage1;%path%
```

`PATH` names the directory with its full path, so that the tools are found from any directory. **`%path%`** keeps the directories that `PATH` already held, after the new one.

Then run `BUILD.BAT` again. It prints the same lines as the first time, and writes the two programs into `BUILD` once more, over the first ones (figure 35.3).

The programs of the second build are the same as those in `STAGE1`, byte for byte: the tools that the first build made make themselves again, exactly. If you keep a copy of the programs that `BUILD` held before the first build, the programs of the release, they are the same as well.

MSX-DOS has no command that compares two files. To compare them, copy them to another computer and compare them there.

When you have finished, take `STAGE1` out of `PATH` again, or start a new session, so that the installed tools are the ones that run.

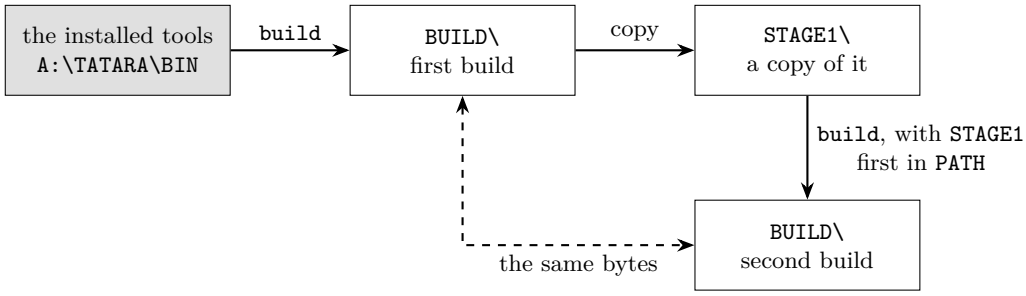


Figure 35.3: Building the tools twice: the second time with the tools that the first build made.

APPENDIX A

Directive reference

A.1 The directives by purpose 279
A.2 The directives in alphabetical order 282
A.3 Labels on directive lines 288
A.4 Other names. 289

This appendix lists every directive that TATARA accepts. Section A.1 groups them by what they are for; section A.2 gives each one's form and a short description, in alphabetical order; section A.3 says what happens to a label written on a directive line; and section A.4 lists the other names that some directives have. Each entry names the section of the manual that describes the directive in full, with examples.

In the forms, words in *italics* stand for something you write, and parts in square brackets can be left out. Directives can be written in upper or lower case.

A.1 The directives by purpose

A.1.1 Data

Directive	What it does	Section
`DB`	stores bytes and strings	11.1
`DW`	stores words, low byte first	11.3
`DC`	stores a string with bit 7 of its last character set	11.4
`DS`	reserves space, without storing anything	10.4

A.1.2 Names

Directive	What it does	Section
`EQU`	gives a name a value, once	12.3
`DEFL`	gives a name a value that can be changed later	12.4
`PUBLIC`	offers names to other modules	12.5
`EXTRN`	uses names that another module defines	12.5

A.1.3 Segments and the location counter

Directive	What it does	Section
`CSEG`	selects the code segment, or a named one	10.2, 10.5
`DSEG`	selects the data segment, or a named one	10.2, 10.5
`ASEG`	selects the absolute segment	10.2
`GROUP`	starts a group in a transient data segment	10.6
`ORG`	sets the location counter	10.3

A.1.4 Files and the end of the source

Directive	What it does	Section
`INCLUDE`	reads another file at this point	13.1
`END`	ends the source, and can name the start address	12.6

A.1.5 Conditional assembly

Directive	What it does	Section
`IF`	assembles its lines if a value is not zero	15.1
`IFE`	assembles its lines if a value is zero	15.1
`IFDEF`	assembles its lines if a name is defined	15.3
`IFNDEF`	assembles its lines if a name is not defined	15.3
`IF1`	assembles its lines on the first pass only	15.4
`IF2`	assembles its lines on the second pass only	15.4
`IFB`	assembles its lines if a text is blank	15.5
`IFNB`	assembles its lines if a text is not blank	15.5
`IFIDN`	assembles its lines if two texts are the same	15.5
`IFDIF`	assembles its lines if two texts differ	15.5
`ELSE`	starts the lines to assemble when the condition is false	15.1
`ENDIF`	ends a conditional	15.1

A.1.6 Macros and repeat blocks

Directive	What it does	Section
`MACRO`	starts the definition of a macro	16.1
`ENDM`	ends a macro or a repeat block	16.1
`LOCAL`	gives names a new spelling in each expansion	16.5
`EXITM`	ends an expansion early	16.6
`REPT`	repeats a block a number of times	17.1
`IRP`	repeats a block once for each item of a list	17.2
`IRPC`	repeats a block once for each character of a text	17.3

A.1.7 The listing

Directive	What it does	Section
`TITLE`	sets the title of the page headings	18.2
`SUBTTL`	sets the subtitle of the page headings	18.2
`PAGE`	starts a new page, and can set the page length	18.2
`.XLIST`	turns the listing off	18.3
`.LIST`	turns the listing back on	18.3
`.XALL`	lists the lines of an expansion that produce bytes	18.4
`.LALL`	lists every line of an expansion	18.4
`.SALL`	lists no line of an expansion	18.4
`.LFCOND`	lists the lines a conditional leaves out	18.5
`.SFCOND`	leaves those lines out of the listing	18.5
`.TFCOND`	changes from one of those two to the other	18.5

A.2 The directives in alphabetical order

The directives whose names begin with a dot, which all control the listing, follow the others in a group of their own (section A.2.1). The other names of a directive are listed in their place, and point to the directive they stand for.

`ASEG`

`aseg`
Selects the absolute segment, whose addresses are exactly the ones written: TANREN does not move it. Each segment keeps its own location counter, so a later `ASEG` continues where the last one stopped. Section 10.2.

`COND` The same as `IF`.

`CSEG`

`cseg [`*`name`*`]`
Selects the code segment, or, with a name, the named code segment of that name. A source starts in the code segment. Every `CSEG` with the same name adds to the same segment, and TANREN joins the segments of the same name from every module. Sections 10.2 and 10.5.

DB

db *item*[,*item*…]
Stores each item: an expression as one byte, its low byte, which must be absolute; a string as one byte for each character. A string of one or two characters followed by an operator is a character constant in an expression instead. Sections 11.1 and 11.2.

DC

dc *string*
Stores one string, as DB does, with bit 7 of its last character set. The string must not be empty. Section 11.4.

DEFB The same as DB.

DEFL

name defl *expression*
Gives a name a value, which a later DEFL of the same name can change. The name is written without a colon, and cannot be a label or an EQU name too. TATARA has no SET: in Z80 code, set is the instruction. Section 12.4.

DEFM The same as DB.

DEFS The same as DS.

DEFW The same as DW.

DS

ds *count*
Moves the location counter on by *count* bytes, so reserving space, and stores nothing. The count must be absolute and known on the first pass. There is no second operand. Section 10.4.

DSEG

dseg [*name*[,transient]]
Selects the data segment, or, with a name, the named data segment of that name. With transient after the name, the segment is a transient data segment, whose variables are divided into groups with GROUP. Sections 10.2, 10.5 and 10.6.

DW

dw *expression*[,*expression*…]
Stores each expression as a word, low byte first. The value can be an address that TANREN finishes, such as a label or an external name. A string of one or two characters is a character constant; a longer one is an error. Section 11.3.

ELSE

`else`

Starts the lines of a conditional that are assembled when its condition is false. It belongs to the innermost conditional that is still open, and is optional. Section 15.1.

END

`end [`*`address`*`]`

Ends the source: TATARA reads nothing after it, not even the rest of a file that included this one. The address, if there is one, is the start address of the program, which TANREN shows as `entry`. A source without `END` ends where its file ends. Sections 12.6 and 13.7.

ENDIF

`endif`

Ends the innermost conditional that is still open. Section 15.1.

ENDM

`endm`

Ends the body of a macro, or the block of `REPT`, `IRP` or `IRPC`. Each `ENDM` belongs to the innermost one that is open. Sections 16.1 and 17.1.

ENTRY The same as `PUBLIC`.

EQU

`name` `equ` *`expression`*

Gives a name a value, which cannot be changed. The name is written without a colon: with one, TATARA takes it for a label and stops. Section 12.3.

EXITM

`exitm`

Ends the expansion of a macro, or the rounds of a repeat block, at once. It is used inside a conditional, and closes the conditionals it leaves open. Sections 16.6 and 17.4.

EXT The same as `EXTRN`.

EXTRN

`extrn` *`name`*`[,`*`name`*`…]`

Declares names that another module defines. TANREN finishes every value that uses them, so they can go only where two bytes hold the value. A name declared `EXTRN` cannot be defined in the same module. Section 12.5.

GROUP

`group` *`name`*

Starts a group in a transient data segment. Every group of the segment starts

at its beginning, and the segment is as large as its largest group. `GROUP` is allowed only in a transient data segment, and takes no label. Group names are shared between modules. Section 10.6.

IF

`if` *`expression`*
Assembles the lines up to `ELSE` or `ENDIF` if the value is not zero. The expression must use only names already defined. Conditionals can be 16 deep. Sections 15.1, 15.2 and 15.6.

IF1

`if1`
Assembles its lines on the first pass only. A label or a byte inside it makes the two passes differ, which is an error. Section 15.4.

IF2

`if2`
Assembles its lines on the second pass only, with the same restriction as `IF1`. Section 15.4.

IFB

`ifb <`*`text`*`>`
Assembles its lines if the text between the angle brackets is blank. Section 15.5.

IFDEF

`ifdef` *`name`*
Assembles its lines if the name is defined. A name declared `EXTRN` counts as defined. The name should be defined above the line, or the answer differs between the passes. Section 15.3.

IFDIF

`ifdif <`*`text`*`>,<`*`text`*`>`
Assembles its lines if the two texts differ. The comparison is exact, upper and lower case included. Section 15.5.

IFE

`ife` *`expression`*
Assembles its lines if the value is zero. Otherwise as `IF`. Section 15.1.

IFF The same as `IFE`.

IFIDN

`ifidn <`*`text`*`>,<`*`text`*`>`
Assembles its lines if the two texts are the same, upper and lower case included. Section 15.5.

IFNB

`ifnb <`*`text`*`>`
Assembles its lines if the text is not blank. Section 15.5.

IFNDEF

`ifndef` *name*
Assembles its lines if the name is not defined. Otherwise as `IFDEF`. Section 15.3.

IFT The same as `IF`.

INCLUDE

`include` *filename*
Reads the file at this point, as if its lines were written there. The name is used as it is written, without quotation marks, and nothing is added to it. TATARA looks for the file beside the file that includes it, then in the current directory, then in each directory of the `TATARA` environment variable. Include files and macro expansions can be 16 deep. Sections 13.1 and 13.2.

IRP

`irp` *parameter*`,<`*item*`[,`*item*`…]>`
Assembles the block up to `ENDM` once for each item, with the parameter replaced by the item. The items follow the rules of a macro's arguments, and the angle brackets can be left out. Section 17.2.

IRPC

`irpc` *parameter*`,`*text*
Assembles the block up to `ENDM` once for each character of the text. Angle brackets are needed only to keep a space in the text. Section 17.3.

LOCAL

`local` *name*`[,`*name*`…]`
In a macro or a repeat block, replaces each name with a new one, `??0000`, `??0001` and so on, in each expansion. `LOCAL` lines must come first in the body. Sections 16.5 and 17.4.

MACRO

name `macro [`*parameter*`[,`*parameter*`…]]`
Starts the definition of a macro, whose body runs to the matching `ENDM`. The name is written without a colon. A macro must be defined before it is used, and can be defined again. Sections 16.1 to 16.7.

ORG

`org` *expression*
Sets the location counter of the current segment: an address in the absolute

segment, an offset in the others. The value must be known on the first pass, and be absolute or an address in the current segment. Section 10.3.

PAGE

`page [n]`

Starts a new page of the listing after its own line. With *n*, from 10 to 255, it also sets the length of a page, page heading included. Section 18.2.

PUBLIC

`public name[,name…]`

Offers names to other modules. The line can come before the definition or after it, but each name must be defined somewhere in the module. Labels, `EQU` names and `DEFL` names can be public. A label written with two colons is public as well. Section 12.5.

REPT

`rept count`

Assembles the block up to `ENDM` *count* times. The count must be absolute and already known. A count of 0 assembles nothing. Section 17.1.

SUBTTL

`subttl text`

Sets the subtitle, shown under the page heading from the next page on. TATARA keeps its first 28 characters. Section 18.2.

TITLE

`title text`

Sets the title, shown at the left of the page heading. TATARA keeps its first 28 characters. Section 18.2.

A.2.1 The directives that begin with a dot

.LALL

`.lall`

Lists every line of a macro expansion or repeat block from here on, including the lines a conditional leaves out. Section 18.4.

.LFCOND

`.lfcond`

Lists the lines that a conditional leaves out, with no address. This is the default. Section 18.5.

.LIST
`.list`
Turns the listing back on after `.XLIST`. Section 18.3.

.SALL
`.sall`
Lists no line of an expansion from here on. The line that uses the macro is marked +. Section 18.4.

.SFCOND
`.sfcond`
Leaves the lines that a conditional leaves out out of the listing. The `IF`, `ELSE` and `ENDIF` lines are still listed. Section 18.5.

.TFCOND
`.tfcond`
Changes from `.LFCOND` to `.SFCOND`, or back. Section 18.5.

.XALL
`.xall`
Lists the lines of an expansion that produce bytes. This is the default. Section 18.4.

.XLIST
`.xlist`
Turns the listing off. The lines after it are assembled as usual, but not listed. Section 18.3.

A.3 Labels on directive lines

A label can be written on most directive lines, as on any other line (section 6.3). It takes the value of the location counter when TATARA reaches the line, in the segment in force before the line does anything. So a label on `ORG` has the old value of the location counter, not the new one; a label on a segment directive has the address in the segment that was in force until that line; and a label on `INCLUDE` has the address before the bytes of the file.

A few directives are different:

- `EQU`, `DEFL` and `MACRO` take a name, not a label. With a colon after the name, TATARA stops with `EQU, DEFL and MACRO take a name, not a label.`

- `GROUP` takes no label. A label on it stops TATARA with `GROUP needs a name, no label, and a transient DSEG.`
- On `IF`, `ELSE` and `ENDIF`, the label is defined only if TATARA is assembling when it reaches the line (section 15.6).
- On `REPT`, `IRP` and `IRPC`, the label has the address of the start of the first round.
- On `EXITM`, the label has the address that the expansion has reached.
- On `LOCAL` and `ENDM` in a macro, the label is ignored, without a message. TATARA reads these two lines when it stores the body of the macro, and never assembles them, so there is no location counter for the label to take. M80 does the same.

The safest place for a label is a line of its own, or a line that produces bytes.

A.4 Other names

Some directives have a second name, which TATARA accepts so that sources written for M80 assemble unchanged (`*EJECT` and `$EJECT` only in column 1):

Other name	Same as
`COND`, `IFT`	`IF`
`DEFB`, `DEFM`	`DB`
`DEFS`	`DS`
`DEFW`	`DW`
`ENTRY`	`PUBLIC`
`EXT`	`EXTRN`
`IFF`	`IFE`
`*EJECT`, `$EJECT`	`PAGE`

The directives of M80 that TATARA does not accept are listed in appendix J.

APPENDIX B

Instruction summary

B.1 How to read the tables 292
B.2 8-bit loads 293
B.3 16-bit loads, PUSH and POP 294
B.4 Exchange, block transfer and search 295
B.5 8-bit arithmetic and logic 296
B.6 General purpose and CPU control 297
B.7 16-bit arithmetic 298
B.8 Rotate and shift 299
B.9 Bit set, reset and test 301
B.10 Jumps 301
B.11 Calls, returns and restarts 302
B.12 Input and output 303
B.13 The undocumented instructions 304
B.14 The R800 multiplications 305
B.15 Forms that are not accepted 305

This appendix lists every instruction form that TATARA accepts, grouped by class as in Zilog's manual, with the bytes TATARA assembles for it and a few words on what it does. Chapter 9 describes how operands are written. What each instruction does in detail, which flags it changes and how long it takes are in the book or course you learn Z80 programming from.

TATARA accepts the documented Z80 instructions, the undocumented ones of section 9.5, and the two multiplications of the R800.

B.1 How to read the tables

Each table has three columns: the form of the instruction, as you write it; the bytes TATARA writes for it, in hexadecimal; and what the instruction does. In the first two columns, a word in *italics* stands for something that changes from one use to the next:

Word	What it stands for
r, *r′*	one of the registers B, C, D, E, H, L and A
n	a byte: any expression, of which the low byte is used
nn	a word: an address or a 16-bit value
lo hi	the two bytes of *nn*, the low byte first
d	the displacement of an indexed operand, from −128 to 127
e	the target of a relative jump; the byte TATARA writes is the distance from the end of the instruction to the target, from −128 to 127
b	a bit number, from 0 to 7
x, *x′*	an index half (section B.13)

Some instructions have a different byte for each register they use. `ld r,r'`, which copies one register into another, comes in 49 versions, from `ld b,b` to `ld a,a`. Instead of a row for each, the table gives a rule for working the byte out. Each register has a number, its *register code*:

B	C	D	E	H	L	(HL)	A
0	1	2	3	4	5	6	7

In the bytes column, `8`*r* means 8 times the code of *r*, and the numbers are added in hexadecimal. The row for `ld r,r'` says `40+8`*r*`+`*r*`'`. For `ld d,b`, the code of D is 2 and the code of B is 0, so the byte is 40h + 8×2 + 0 = 40h + 10h = **50h**. In the same way, `add a,r` is `80+`*r*, so `add a,e` is 80h + 3 = 83h, and `bit b,r` is `CB 40+8`*b*`+`*r*, so `bit 7,a` is `CB`, then 40h + 8×7 + 7 = 40h + 38h + 7 = 7Fh.

Where an instruction has only a few versions, such as the register pairs of `push` or the conditions of `jp`, each one has a row of its own, with its bytes written out.

The instructions that use IX or IY are the ones that use HL, with a prefix byte in front: `DD` for IX and `FD` for IY. In the forms with *d*, the displacement byte follows the opcode; in those that start `DD CB` or `FD CB`, it comes before the last byte.

B.2 8-bit loads

Form	Bytes	What it does
`ld r,r'`	`40+8r+r'`	copies *r′* into *r*
`ld r,n`	`06+8r n`	loads *n* into *r*
`ld r,(hl)`	`46+8r`	loads the byte at the address in HL
`ld r,(ix+d)`	`DD 46+8r d`	loads the byte at IX+*d*
`ld r,(iy+d)`	`FD 46+8r d`	loads the byte at IY+*d*
`ld (hl),r`	`70+r`	stores *r* at the address in HL
`ld (ix+d),r`	`DD 70+r d`	stores *r* at IX+*d*
`ld (iy+d),r`	`FD 70+r d`	stores *r* at IY+*d*
`ld (hl),n`	`36 n`	stores *n* at the address in HL
`ld (ix+d),n`	`DD 36 d n`	stores *n* at IX+*d*
`ld (iy+d),n`	`FD 36 d n`	stores *n* at IY+*d*
`ld a,(bc)`	`0A`	loads A from the address in BC
`ld a,(de)`	`1A`	loads A from the address in DE
`ld a,(nn)`	`3A lo hi`	loads A from address *nn*
`ld (bc),a`	`02`	stores A at the address in BC
`ld (de),a`	`12`	stores A at the address in DE
`ld (nn),a`	`32 lo hi`	stores A at address *nn*
`ld a,i`	`ED 57`	copies the interrupt vector register into A
`ld a,r`	`ED 5F`	copies the refresh register into A
`ld i,a`	`ED 47`	copies A into the interrupt vector register
`ld r,a`	`ED 4F`	copies A into the refresh register

B.3 16-bit loads, PUSH and POP

Form	Bytes	What it does
`ld bc,`*nn*	`01` *lo hi*	loads *nn* into BC
`ld de,`*nn*	`11` *lo hi*	loads *nn* into DE
`ld hl,`*nn*	`21` *lo hi*	loads *nn* into HL
`ld sp,`*nn*	`31` *lo hi*	loads *nn* into SP
`ld ix,`*nn*	`DD 21` *lo hi*	loads *nn* into IX
`ld iy,`*nn*	`FD 21` *lo hi*	loads *nn* into IY
`ld bc,(`*nn*`)`	`ED 4B` *lo hi*	loads BC from address *nn*
`ld de,(`*nn*`)`	`ED 5B` *lo hi*	loads DE from address *nn*
`ld hl,(`*nn*`)`	`2A` *lo hi*	loads HL from address *nn*
`ld sp,(`*nn*`)`	`ED 7B` *lo hi*	loads SP from address *nn*
`ld ix,(`*nn*`)`	`DD 2A` *lo hi*	loads IX from address *nn*
`ld iy,(`*nn*`)`	`FD 2A` *lo hi*	loads IY from address *nn*
`ld (`*nn*`),bc`	`ED 43` *lo hi*	stores BC at address *nn*
`ld (`*nn*`),de`	`ED 53` *lo hi*	stores DE at address *nn*
`ld (`*nn*`),hl`	`22` *lo hi*	stores HL at address *nn*
`ld (`*nn*`),sp`	`ED 73` *lo hi*	stores SP at address *nn*
`ld (`*nn*`),ix`	`DD 22` *lo hi*	stores IX at address *nn*
`ld (`*nn*`),iy`	`FD 22` *lo hi*	stores IY at address *nn*
`ld sp,hl`	`F9`	copies HL into SP
`ld sp,ix`	`DD F9`	copies IX into SP
`ld sp,iy`	`FD F9`	copies IY into SP
`push bc`	`C5`	puts BC on the stack
`push de`	`D5`	puts DE on the stack
`push hl`	`E5`	puts HL on the stack
`push af`	`F5`	puts AF on the stack
`push ix`	`DD E5`	puts IX on the stack
`push iy`	`FD E5`	puts IY on the stack
`pop bc`	`C1`	takes BC from the stack
`pop de`	`D1`	takes DE from the stack
`pop hl`	`E1`	takes HL from the stack

Form	Bytes	What it does
`pop af`	`F1`	takes AF from the stack
`pop ix`	`DD E1`	takes IX from the stack
`pop iy`	`FD E1`	takes IY from the stack

B.4 Exchange, block transfer and search

Form	Bytes	What it does
`ex de,hl`	`EB`	exchanges DE and HL
`ex af,af'`	`08`	exchanges AF and the second AF
`exx`	`D9`	exchanges BC, DE and HL with the second set
`ex (sp),hl`	`E3`	exchanges HL and the word at the top of the stack
`ex (sp),ix`	`DD E3`	exchanges IX and the word at the top of the stack
`ex (sp),iy`	`FD E3`	exchanges IY and the word at the top of the stack
`ldi`	`ED A0`	copies a byte from (HL) to (DE), then moves on
`ldir`	`ED B0`	repeats LDI until BC is 0
`ldd`	`ED A8`	copies a byte from (HL) to (DE), then moves back
`lddr`	`ED B8`	repeats LDD until BC is 0
`cpi`	`ED A1`	compares A with (HL), then moves on
`cpir`	`ED B1`	repeats CPI until a match or BC is 0
`cpd`	`ED A9`	compares A with (HL), then moves back
`cpdr`	`ED B9`	repeats CPD until a match or BC is 0

B.5 8-bit arithmetic and logic

Form	Bytes	What it does
`add a,`*r*	`80+`*r*	adds *r* to A
`add a,`*n*	`C6` *n*	adds *n* to A
`add a,(hl)`	`86`	adds the byte at (HL) to A
`add a,(ix+`*d*`)`	`DD 86` *d*	adds the byte at IX+*d* to A
`add a,(iy+`*d*`)`	`FD 86` *d*	adds the byte at IY+*d* to A
`adc a,`*r*	`88+`*r*	adds *r* and the carry to A
`adc a,`*n*	`CE` *n*	adds *n* and the carry to A
`adc a,(hl)`	`8E`	adds the byte at (HL) and the carry to A
`adc a,(ix+`*d*`)`	`DD 8E` *d*	adds the byte at IX+*d* and the carry to A
`adc a,(iy+`*d*`)`	`FD 8E` *d*	adds the byte at IY+*d* and the carry to A
`sub` *r*	`90+`*r*	subtracts *r* from A
`sub` *n*	`D6` *n*	subtracts *n* from A
`sub (hl)`	`96`	subtracts the byte at (HL) from A
`sub (ix+`*d*`)`	`DD 96` *d*	subtracts the byte at IX+*d* from A
`sub (iy+`*d*`)`	`FD 96` *d*	subtracts the byte at IY+*d* from A
`sbc a,`*r*	`98+`*r*	subtracts *r* and the carry from A
`sbc a,`*n*	`DE` *n*	subtracts *n* and the carry from A
`sbc a,(hl)`	`9E`	subtracts the byte at (HL) and the carry from A
`sbc a,(ix+`*d*`)`	`DD 9E` *d*	subtracts the byte at IX+*d* and the carry from A
`sbc a,(iy+`*d*`)`	`FD 9E` *d*	subtracts the byte at IY+*d* and the carry from A
`and` *r*	`A0+`*r*	A AND *r*
`and` *n*	`E6` *n*	A AND *n*
`and (hl)`	`A6`	A AND the byte at (HL)
`and (ix+`*d*`)`	`DD A6` *d*	A AND the byte at IX+*d*
`and (iy+`*d*`)`	`FD A6` *d*	A AND the byte at IY+*d*
`xor` *r*	`A8+`*r*	A XOR *r*
`xor` *n*	`EE` *n*	A XOR *n*

Form	Bytes	What it does
`xor (hl)`	`AE`	A XOR the byte at (HL)
`xor (ix+d)`	`DD AE d`	A XOR the byte at IX+*d*
`xor (iy+d)`	`FD AE d`	A XOR the byte at IY+*d*
`or r`	`B0+r`	A OR *r*
`or n`	`F6 n`	A OR *n*
`or (hl)`	`B6`	A OR the byte at (HL)
`or (ix+d)`	`DD B6 d`	A OR the byte at IX+*d*
`or (iy+d)`	`FD B6 d`	A OR the byte at IY+*d*
`cp r`	`B8+r`	compares A with *r*
`cp n`	`FE n`	compares A with *n*
`cp (hl)`	`BE`	compares A with the byte at (HL)
`cp (ix+d)`	`DD BE d`	compares A with the byte at IX+*d*
`cp (iy+d)`	`FD BE d`	compares A with the byte at IY+*d*
`inc r`	`04+8r`	adds 1 to *r*
`inc (hl)`	`34`	adds 1 to the byte at (HL)
`inc (ix+d)`	`DD 34 d`	adds 1 to the byte at IX+*d*
`inc (iy+d)`	`FD 34 d`	adds 1 to the byte at IY+*d*
`dec r`	`05+8r`	subtracts 1 from *r*
`dec (hl)`	`35`	subtracts 1 from the byte at (HL)
`dec (ix+d)`	`DD 35 d`	subtracts 1 from the byte at IX+*d*
`dec (iy+d)`	`FD 35 d`	subtracts 1 from the byte at IY+*d*

B.6 General purpose and CPU control

Form	Bytes	What it does
`daa`	`27`	adjusts A after a BCD addition or subtraction
`cpl`	`2F`	inverts every bit of A
`neg`	`ED 44`	negates A
`ccf`	`3F`	inverts the carry flag

Form	Bytes	What it does
`scf`	`37`	sets the carry flag
`nop`	`00`	does nothing
`halt`	`76`	waits for an interrupt
`di`	`F3`	disables interrupts
`ei`	`FB`	enables interrupts
`im 0`	`ED 46`	selects interrupt mode 0
`im 1`	`ED 56`	selects interrupt mode 1
`im 2`	`ED 5E`	selects interrupt mode 2

B.7 16-bit arithmetic

Form	Bytes	What it does
`add hl,bc`	`09`	adds BC to HL
`add hl,de`	`19`	adds DE to HL
`add hl,hl`	`29`	adds HL to HL
`add hl,sp`	`39`	adds SP to HL
`adc hl,bc`	`ED 4A`	adds BC and the carry to HL
`adc hl,de`	`ED 5A`	adds DE and the carry to HL
`adc hl,hl`	`ED 6A`	adds HL and the carry to HL
`adc hl,sp`	`ED 7A`	adds SP and the carry to HL
`sbc hl,bc`	`ED 42`	subtracts BC and the carry from HL
`sbc hl,de`	`ED 52`	subtracts DE and the carry from HL
`sbc hl,hl`	`ED 62`	subtracts HL and the carry from HL
`sbc hl,sp`	`ED 72`	subtracts SP and the carry from HL
`add ix,bc`	`DD 09`	adds BC to IX
`add ix,de`	`DD 19`	adds DE to IX
`add ix,ix`	`DD 29`	adds IX to IX
`add ix,sp`	`DD 39`	adds SP to IX
`add iy,bc`	`FD 09`	adds BC to IY

Form	Bytes	What it does
`add iy,de`	`FD 19`	adds DE to IY
`add iy,iy`	`FD 29`	adds IY to IY
`add iy,sp`	`FD 39`	adds SP to IY
`inc bc`	`03`	adds 1 to BC
`inc de`	`13`	adds 1 to DE
`inc hl`	`23`	adds 1 to HL
`inc sp`	`33`	adds 1 to SP
`inc ix`	`DD 23`	adds 1 to IX
`inc iy`	`FD 23`	adds 1 to IY
`dec bc`	`0B`	subtracts 1 from BC
`dec de`	`1B`	subtracts 1 from DE
`dec hl`	`2B`	subtracts 1 from HL
`dec sp`	`3B`	subtracts 1 from SP
`dec ix`	`DD 2B`	subtracts 1 from IX
`dec iy`	`FD 2B`	subtracts 1 from IY

B.8 Rotate and shift

Form	Bytes	What it does
`rlca`	`07`	rotates A left
`rla`	`17`	rotates A left through the carry
`rrca`	`0F`	rotates A right
`rra`	`1F`	rotates A right through the carry
`rlc` *r*	`CB 00+`*r*	rotates *r* left
`rlc (hl)`	`CB 06`	rotates the byte at (HL) left
`rlc (ix+`*d*`)`	`DD CB` *d* `06`	rotates the byte at IX+*d* left
`rlc (iy+`*d*`)`	`FD CB` *d* `06`	rotates the byte at IY+*d* left
`rl` *r*	`CB 10+`*r*	rotates *r* left through the carry
`rl (hl)`	`CB 16`	rotates the byte at (HL) left through the carry

Form	Bytes	What it does
`rl (ix+d)`	`DD CB d 16`	rotates the byte at IX+*d* left through the carry
`rl (iy+d)`	`FD CB d 16`	rotates the byte at IY+*d* left through the carry
`rrc r`	`CB 08+r`	rotates *r* right
`rrc (hl)`	`CB 0E`	rotates the byte at (HL) right
`rrc (ix+d)`	`DD CB d 0E`	rotates the byte at IX+*d* right
`rrc (iy+d)`	`FD CB d 0E`	rotates the byte at IY+*d* right
`rr r`	`CB 18+r`	rotates *r* right through the carry
`rr (hl)`	`CB 1E`	rotates the byte at (HL) right through the carry
`rr (ix+d)`	`DD CB d 1E`	rotates the byte at IX+*d* right through the carry
`rr (iy+d)`	`FD CB d 1E`	rotates the byte at IY+*d* right through the carry
`sla r`	`CB 20+r`	shifts *r* left
`sla (hl)`	`CB 26`	shifts the byte at (HL) left
`sla (ix+d)`	`DD CB d 26`	shifts the byte at IX+*d* left
`sla (iy+d)`	`FD CB d 26`	shifts the byte at IY+*d* left
`sra r`	`CB 28+r`	shifts *r* right, keeping bit 7
`sra (hl)`	`CB 2E`	shifts the byte at (HL) right, keeping bit 7
`sra (ix+d)`	`DD CB d 2E`	shifts the byte at IX+*d* right, keeping bit 7
`sra (iy+d)`	`FD CB d 2E`	shifts the byte at IY+*d* right, keeping bit 7
`srl r`	`CB 38+r`	shifts *r* right, with 0 into bit 7
`srl (hl)`	`CB 3E`	shifts the byte at (HL) right, with 0 into bit 7
`srl (ix+d)`	`DD CB d 3E`	shifts the byte at IX+*d* right, with 0 into bit 7
`srl (iy+d)`	`FD CB d 3E`	shifts the byte at IY+*d* right, with 0 into bit 7
`rld`	`ED 6F`	rotates a digit left between A and (HL)

Form	Bytes	What it does
`rrd`	`ED 67`	rotates a digit right between A and (HL)

B.9 Bit set, reset and test

Form	Bytes	What it does
`bit b,r`	`CB 40+8b+r`	tests bit *b* of *r*
`bit b,(hl)`	`CB 46+8b`	tests bit *b* of the byte at (HL)
`bit b,(ix+d)`	`DD CB d 46+8b`	tests bit *b* of the byte at IX+*d*
`bit b,(iy+d)`	`FD CB d 46+8b`	tests bit *b* of the byte at IY+*d*
`set b,r`	`CB C0+8b+r`	sets bit *b* of *r*
`set b,(hl)`	`CB C6+8b`	sets bit *b* of the byte at (HL)
`set b,(ix+d)`	`DD CB d C6+8b`	sets bit *b* of the byte at IX+*d*
`set b,(iy+d)`	`FD CB d C6+8b`	sets bit *b* of the byte at IY+*d*
`res b,r`	`CB 80+8b+r`	clears bit *b* of *r*
`res b,(hl)`	`CB 86+8b`	clears bit *b* of the byte at (HL)
`res b,(ix+d)`	`DD CB d 86+8b`	clears bit *b* of the byte at IX+*d*
`res b,(iy+d)`	`FD CB d 86+8b`	clears bit *b* of the byte at IY+*d*

B.10 Jumps

Form	Bytes	What it does
`jp nn`	`C3 lo hi`	jumps to *nn*
`jp nz,nn`	`C2 lo hi`	jumps to *nn* if not zero
`jp z,nn`	`CA lo hi`	jumps to *nn* if zero
`jp nc,nn`	`D2 lo hi`	jumps to *nn* if no carry
`jp c,nn`	`DA lo hi`	jumps to *nn* if carry

Form	Bytes	What it does
`jp po,`*nn*	`E2` *lo hi*	jumps to *nn* if parity odd
`jp pe,`*nn*	`EA` *lo hi*	jumps to *nn* if parity even
`jp p,`*nn*	`F2` *lo hi*	jumps to *nn* if positive
`jp m,`*nn*	`FA` *lo hi*	jumps to *nn* if negative
`jr` *e*	`18` *e*	jumps to *e*
`jr nz,`*e*	`20` *e*	jumps to *e* if not zero
`jr z,`*e*	`28` *e*	jumps to *e* if zero
`jr nc,`*e*	`30` *e*	jumps to *e* if no carry
`jr c,`*e*	`38` *e*	jumps to *e* if carry
`jp (hl)`	`E9`	jumps to the address in HL
`jp (ix)`	`DD E9`	jumps to the address in IX
`jp (iy)`	`FD E9`	jumps to the address in IY
`djnz` *e*	`10` *e*	subtracts 1 from B and jumps to *e* if B is not 0

B.11 Calls, returns and restarts

Form	Bytes	What it does
`call` *nn*	`CD` *lo hi*	calls the routine at *nn*
`call nz,`*nn*	`C4` *lo hi*	calls the routine at *nn* if not zero
`call z,`*nn*	`CC` *lo hi*	calls the routine at *nn* if zero
`call nc,`*nn*	`D4` *lo hi*	calls the routine at *nn* if no carry
`call c,`*nn*	`DC` *lo hi*	calls the routine at *nn* if carry
`call po,`*nn*	`E4` *lo hi*	calls the routine at *nn* if parity odd
`call pe,`*nn*	`EC` *lo hi*	calls the routine at *nn* if parity even
`call p,`*nn*	`F4` *lo hi*	calls the routine at *nn* if positive
`call m,`*nn*	`FC` *lo hi*	calls the routine at *nn* if negative
`ret`	`C9`	returns from a routine
`ret nz`	`C0`	returns if not zero
`ret z`	`C8`	returns if zero

Form	Bytes	What it does
`ret nc`	`D0`	returns if no carry
`ret c`	`D8`	returns if carry
`ret po`	`E0`	returns if parity odd
`ret pe`	`E8`	returns if parity even
`ret p`	`F0`	returns if positive
`ret m`	`F8`	returns if negative
`reti`	`ED 4D`	returns from an interrupt routine
`retn`	`ED 45`	returns from a non-maskable interrupt routine
`rst 00h`	`C7`	calls the routine at 00h
`rst 08h`	`CF`	calls the routine at 08h
`rst 10h`	`D7`	calls the routine at 10h
`rst 18h`	`DF`	calls the routine at 18h
`rst 20h`	`E7`	calls the routine at 20h
`rst 28h`	`EF`	calls the routine at 28h
`rst 30h`	`F7`	calls the routine at 30h
`rst 38h`	`FF`	calls the routine at 38h

B.12 Input and output

Form	Bytes	What it does
`in a,(`*n*`)`	`DB` *n*	reads port *n* into A
`in` *r*`,(c)`	`ED 40+8`*r*	reads the port in C into *r*
`ini`	`ED A2`	reads the port in C into (HL), then moves on
`inir`	`ED B2`	repeats INI until B is 0
`ind`	`ED AA`	reads the port in C into (HL), then moves back
`indr`	`ED BA`	repeats IND until B is 0
`out (`*n*`),a`	`D3` *n*	writes A to port *n*

Form	Bytes	What it does
`out (c),r`	`ED 41+8r`	writes *r* to the port in C
`outi`	`ED A3`	writes (HL) to the port in C, then moves on
`otir`	`ED B3`	repeats OUTI until B is 0
`outd`	`ED AB`	writes (HL) to the port in C, then moves back
`otdr`	`ED BB`	repeats OUTD until B is 0

B.13 The undocumented instructions

These are the undocumented instructions of section 9.5. In the `SLL` rows, *r* is any of the seven registers. In the other rows, *x* and *x′* are `IXH` or `IXL`, whose codes are 4 and 5, the codes of H and L; *r* is B, C, D, E or A; and *x′* is a half of the same register as *x*. The rows give the bytes for IX. For `IYH` and `IYL`, `FD` takes the place of `DD`: `ld iyl,a` is `FD 6F`.

Form	Bytes	What it does
`sll r`	`CB 30+r`	shifts *r* left; bit 0 becomes 1
`sll (hl)`	`CB 36`	the same with the byte at the address in HL
`sll (ix+d)`	`DD CB d 36`	the same with the byte at IX+*d*
`sll (iy+d)`	`FD CB d 36`	the same with the byte at IY+*d*
`ld x,r`	`DD 40+8x+r`	copies *r* into *x*
`ld r,x`	`DD 40+8r+x`	copies *x* into *r*
`ld x,x'`	`DD 40+8x+x'`	copies one half of IX into the other, or into itself
`ld x,n`	`DD 06+8x n`	loads *n* into *x*
`inc x`	`DD 04+8x`	adds 1 to *x*
`dec x`	`DD 05+8x`	subtracts 1 from *x*
`add a,x`	`DD 80+x`	adds *x* to A
`adc a,x`	`DD 88+x`	adds *x* and the carry to A
`sub x`	`DD 90+x`	subtracts *x* from A

Form	Bytes	What it does
`sbc a,`*x*	`DD 98+`*x*	subtracts *x* and the carry from A
`and` *x*	`DD A0+`*x*	A AND *x*
`xor` *x*	`DD A8+`*x*	A XOR *x*
`or` *x*	`DD B0+`*x*	A OR *x*
`cp` *x*	`DD B8+`*x*	compares A with *x*
`in f,(c)`	`ED 70`	reads the port in C and sets the flags; A is unchanged
`out (c),0`	`ED 71`	writes 0 or 0FFh to the port in C (section 9.5)

B.14 The R800 multiplications

Form	Bytes	What it does
`mulub a,`*r*	`ED C1+8`*r*	A times *r*, the result in HL
`muluw hl,bc`	`ED C3`	HL times BC, the result in DE:HL
`muluw hl,sp`	`ED F3`	HL times SP, the result in DE:HL

B.15 Forms that are not accepted

Some forms are accepted by other assemblers, or look as if they should work, but are not in Zilog's list or among the undocumented instructions TATARA accepts. TATARA refuses them (table B.14):

`SUB`, `AND`, `OR`, `XOR` and `CP` always work on A, so Zilog writes them without it; `ADD`, `ADC` and `SBC` have 16-bit forms too, so their 8-bit forms name A. Without brackets, `jp hl` is read as `jp nn`, with `hl` as the name of an address, which is not defined.

The forms with an index half cannot be written because one prefix governs the whole instruction (section 9.5). `rlc (ix+2),b` is one of the undocumented forms that store the result in a register as well as in memory, which TATARA does not accept. `in (c)`, `sl1` and `hx` are other assemblers' spellings of instructions that TATARA writes in one way only.

Written	Message	Write instead
`sub a,b`	`not a form this instruction has.`	`sub b`
`and a,b`	`not a form this instruction has.`	`and b`
`cp a,b`	`not a form this instruction has.`	`cp b`
`add b`	`not a form this instruction has.`	`add a,b`
`ex af,af`	`not a form this instruction has.`	`ex af,af'`
`jp hl`	`undefined symbol in an expression.`	`jp (hl)`
`ld h,ixl`	`not a form this instruction has.`	none
`ld ixh,iyl`	`not a form this instruction has.`	none
`ld ixh,(hl)`	`not a form this instruction has.`	none
`rlc ixh`	`not a form this instruction has.`	none
`rlc (ix+2),b`	`not a form this instruction has.`	`rlc (ix+2)` and `ld b,(ix+2)`
`in (c)`	`not a form this instruction has.`	`in f,(c)`
`sl1 b`	`not a directive, a macro or an instruction.`	`sll b`
`ld a,hx`	`undefined symbol in an expression.`	`ld a,ixh`

Table B.14: Forms that TATARA refuses.

`(IX)` and `(IY)` without a displacement are accepted, and mean `(IX+0)` and `(IY+0)`: `ld a,(ix)` is `DD 7E 00`.

APPENDIX C

Operators and precedence

C.1 The operators 309
C.2 At the edges 311
C.3 Values 311

This appendix gathers what chapter 8 says about expressions into a few tables, for reference: the operators and their precedence, the cases at the edges of the 16-bit range, and the values an expression can be built from.

C.1 The operators

Table C.1 lists every operator, grouped by level of precedence. Level 1 binds the tightest: in an expression with more than one operator, the operators of the lower level are worked out first.

These rules apply to all of them (section 8.5):

- Operators of the same level are worked out from left to right: `10-4-3` is `(10-4)-3`, `1 shl 3 * 2` is 10h, and `1 or 2 xor 3` is `(1 or 2) xor 3`, which is 0.
- Brackets change the order: `2+3*4` is 14, and `(2+3)*4` is 20.
- `HIGH` and `LOW` apply to the value right after them: `high 1234h+1` is `(high 1234h)+1`, which is 13h.
- Negation binds less tightly than `*`, `/` and `MOD`: `-2*3` is `-(2*3)`, and `-3/2` is `-(3/2)`, which is -1.
- `NOT` binds less tightly than the comparisons: `not 1 eq 1` is `not (1 eq 1)`, which is 0.

Level	Operator	What it does	Example and value
1	`HIGH`	high byte	`high 1234h` = 12h
	`LOW`	low byte	`low 1234h` = 34h
2	`*`	multiplication	`6*7` = 42
	`/`	division, unsigned	`17/5` = 3
	`MOD`	remainder of the division, unsigned	`17 mod 5` = 2
	`SHL`	shift left	`1 shl 4` = 10h
	`SHR`	shift right	`8000h shr 15` = 1
3	`-`	negation, before a value	`-1` = 0FFFFh
4	`+`	addition	`2+3` = 5
	`-`	subtraction	`10-4-3` = 3
5	`EQ NE`	equal, not equal	`1 eq 1` = 0FFFFh
	`LT LE`	less than, less or equal	`2 le 1` = 0
	`GT GE`	greater than, greater or equal	`2 gt 1` = 0FFFFh
6	`NOT`	inverts every bit	`not 0` = 0FFFFh
7	`AND`	bitwise and	`0f0h and 3ch` = 30h
8	`OR`	bitwise or	`0f0h or 0fh` = 0FFh
	`XOR`	bitwise exclusive or	`0ffh xor 0fh` = 0F0h

Table C.1: The operators, by level of precedence, tightest first.

- `AND` binds more tightly than `OR` and `XOR`: `2 or 1 and 0` is `2 or (1 and 0)`, which is 2.
- The comparisons give 0FFFFh for true and 0 for false, and compare unsigned numbers: `-1 gt 1` is true.
- The word operators can be written in upper or lower case, and need a space or a bracket on each side: `a and b`, not `aandb`.

These are M80's levels. When in doubt, add brackets: they cost nothing.

C.2 At the edges

Every value is a 16-bit number, from 0 to 65535, and a negative number is stored as its 16-bit complement (section 8.1). Table C.2 shows what the operators give at the edges of that range.

Expression	Value	Why
`65535+1`	0	the value wraps round
`0-1`	0FFFFh	the value wraps round
`17/0`	0FFFFh	division by 0 is not an error
`17 mod 0`	0	nor is the remainder of it
`1 shl 16`	0	every bit is shifted out
`1 shl 0ffffh`	0	likewise
`(-3)/2`	7FFEh	−3 is 0FFFDh, and division is unsigned
`(-7) mod 2`	1	−7 is 0FFF9h, and the remainder is unsigned
`-3/2`	0FFFFh	this is `-(3/2)`: negation comes after `/`
`high -1`	0FFh	−1 is 0FFFFh

Table C.2: Results at the edges of the 16-bit range.

Where a line needs only one byte, such as the operand of `ld a,n` or of `DB`, TATARA uses the low byte of the value and does not check that it fits.

C.3 Values

C.3.1 Numbers, characters and the location counter

A number is decimal unless a letter after it names another base (section 8.2). It must start with a digit, so a hexadecimal number that starts with a letter needs a `0` in front.

Base	Suffix	Example	Value
decimal	none, or `D`	`100`, `100d`	100
hexadecimal	`H`	`100h`, `0ffh`	256, 255
octal	`O` or `Q`	`100o`, `100q`	64
binary	`B`	`100b`	4

`B` and `D` are hexadecimal digits as well as suffixes, so always end a hexadecimal number with its `H`: `10b` is 2.

A character constant is one or two characters between apostrophes or quotation marks, and its value is their code: `'A'` is 41h, and `"AB"` is 4142h, with the first character in the high byte (section 8.3).

`$` on its own is the location counter: the address of the line it is written in, in the current segment (section 8.4).

C.3.2 Kinds of value

An expression is *absolute*, *relocatable* or *external* (section 8.6). An absolute value is complete when TATARA assembles it; a relocatable one is an address in a segment, which TANREN finishes; an external one is a name declared with `EXTRN`. Only these combinations are allowed:

Expression	Result	Example
absolute + relocatable, or the reverse	relocatable	`a1+2`
relocatable − absolute	relocatable	`a1-2`
relocatable − relocatable, same segment	absolute	`a2-a1`
external + absolute, or the reverse	external	`far+2`
external − absolute	external	`far-2`

Every other operator, including negation, `NOT`, `HIGH` and `LOW`, needs absolute values. An external name can appear only once in an expression. A relocatable or external value can go only where two bytes hold it, as in `ld hl,a1` or `DW`; a one-byte operand must be absolute.

APPENDIX D

Error messages

D.1 How a message is laid out 313
D.2 TATARA's messages 314
D.3 TANREN's messages. 318
D.4 TANREN's warnings and notes 320

This appendix lists every message that TATARA and TANREN print: what it means, what to do about it, and the section that shows it with an example. Section D.1 describes how a message is laid out; sections D.2 and D.3 list the messages of each tool in alphabetical order, as they are printed; and section D.4 lists TANREN's warnings and notes, which are not errors.

D.1 How a message is laid out

A TATARA error names the file and the line, then the message after `ERROR:` (section 20.1):

MSX-DOS

```
LERR.AS(5): ERROR: not a form this instruction has.
```

When the line is in an include file, a macro or a repeat block, the lines after the message, the *trail*, show how TATARA got there: `included from FILE(line)` for an include file, and `in NAME, called from FILE(line)` for a macro (section 20.3). A mistake on the command line, or a missing MSX-DOS2 or memory mapper, has no line to name: the message is `ERROR:` and the text, before the banner.

A TANREN error names no line. When the mistake is in one object file, the message begins with the file's name, as in `BAD.TRO: ERROR: not a Tatara object file.`; otherwise it is `ERROR:` and the text (section 27.1).

Both tools print one error and stop (sections 20.2 and 27.2). Correct the mistake and run the tool again to find the next one. `/Q` never hides an error. File names in messages are printed in capitals.

In the lists below, each message is shown as it is printed after `ERROR:`, and a word in *italics* stands for a name that changes. The lists are in alphabetical order of the message as printed, so `a PUBLIC name was never defined.` is under A.

D.2 TATARA's messages

`a bit number must be 0 to 7.`
The bit number of `BIT`, `SET` or `RES` is outside 0 to 7. Section 9.6.

`a JR or DJNZ can only reach -128 to 127.`
The target of a relative jump is too far away. Use `JP`, which can reach any address. Section 9.3.

`a macro name may be at most 64 characters.`
The name before `MACRO` is longer than 64 characters. Section 7.2.

`a name may hold only letters, digits, ? @ . _ and the dollar sign.`
A label holds a character that a name cannot have, such as `*` in `foo*bar`. Section 7.1.

`a PAGE length must be 10 to 255.`
The count after `PAGE` is outside that range. Section 18.2.

`a PUBLIC name was never defined.`
A name declared `PUBLIC` is not defined anywhere in the source. It is found when the object file is written, so without an object file the source is accepted. Section 12.5.

`a segment or group name may be at most 16 characters.`
The name of a segment or group, after `CSEG`, `DSEG` or `GROUP`, is longer than 16 characters. Section 7.2.

`an external symbol may not be used here.`
A name declared `EXTRN` is used where TANREN could not finish it: in `EQU`, twice in one expression, or with an operator other than adding or subtracting a number, as in `2-far`. Section 8.6.

`an index displacement must be -128 to 127.`
The displacement *d* of `(IX+d)` or `(IY+d)` is outside that range. Section 9.2.

`bad ASEG, CSEG or DSEG line.`
A segment directive has an operand it does not take, such as a word after the name other than `TRANSIENT`. Section 10.7.

`bad DB, DW or DC operand.`
: An item is missing, as in `db 1,,2`; a string in `DW` has more than two characters; or `DC` has no string, or an empty one. Section 11.4.

`bad expression.`
: An expression cannot be read: a digit outside its base, a name with a character names cannot have, an operand where none is allowed, or nothing at all where a value is needed. Section 8.2.

`bad macro parameter list.`
: The parameters of `MACRO`, or the parameter and list of `IRP` or `IRPC`, are not names separated by commas. Section 16.9.

`bad PUBLIC or EXTRN list.`
: `PUBLIC` or `EXTRN` has no names, or something that is not a name. Section 12.7.

`cannot create` *name*
: The object file or the listing cannot be made, for example because the directory does not exist or the disk is full or write-protected. Section 5.6.

`cannot open` *name*
: The source, or a file named by `INCLUDE`, was not found. For an include file, the lines after the message show which line asked for it, and TATARA has looked in the three places of section 13.2. Section 13.8.

`cannot write to the output file.`
: Writing the object file or the listing failed, because the disk is full or write-protected. As after any error on the second pass, TATARA deletes the object file. Make room on the disk, or use another one, and assemble again. Section 20.2.

`conditional not closed by ENDIF.`
: An `IF` has no `ENDIF`. The message names the line of the `IF`. Section 15.7.

`conditionals nested too deeply.`
: More than 16 conditionals are open at once. Section 15.7.

`ELSE or ENDIF without a matching IF.`
: `ELSE` or `ENDIF` with no `IF` open, or a second `ELSE` for one `IF`. Section 15.7.

`ENDM without a macro definition.`
: `ENDM` with no `MACRO` or repeat block open. Section 16.9.

`EQU, DEFL and MACRO take a name, not a label.`
: The name before `EQU`, `DEFL` or `MACRO` has a colon after it. Remove the colon. Section 12.3.

`EXITM outside a macro or repeat block.`
: `EXITM` is not inside a macro or a repeat block. Section 16.9.

`GROUP needs a name, no label, and a transient DSEG.`

`GROUP` has no name, has a label, or is not in a transient data segment. Section 10.7.

`IM takes 0, 1 or 2.`
The operand of `IM` is not 0, 1 or 2. Section 9.6.

`INCLUDE without a filename.`
`INCLUDE` has nothing after it. Section 13.8.

`internal - an instruction emitted nothing.`
An internal error: TATARA has found a fault in itself. Please report it (section 20.5). Section 20.5.

`internal - this line emitted more bytes than fit.`
An internal error, as above. Section 20.5.

`IXH, IXL, IYH and IYL are registers, not names.`
A label, `PUBLIC` or `EXTRN` uses one of the four names of the index halves, which are registers since version 1.2.0. M80 allows them as names. Section 9.5.

`LOCAL must come before the macro body.`
A `LOCAL` line comes after another line of the body. Section 16.5.

`macro definition not closed by ENDM.`
A macro or repeat block has no `ENDM`. The message names the line that started it. Section 16.9.

`macro line too long.`
A line of a macro or repeat block is longer than 255 characters once its parameters are replaced by the arguments. The trail names the macro. Section 16.2.

`MACRO without a name.`
`MACRO` has no name before it. Section 16.9.

`not a directive, a macro or an instruction.`
The word in the operation field is none of these. It may be misspelt, an M80 directive that TATARA does not have (appendix J), a macro not yet defined, or a label that does not start in column 1 and has no colon. Section 6.3.

`not a form this instruction has.`
The instruction exists, but not with these operands, as `ld hl,a`. Appendix B lists the forms. Section 9.6.

`only *EJECT and its dollar spelling are control lines.`
A word in column 1 starts with `*` or `$`, which makes the line a control line, as in M80, but it is not `*EJECT` or `$EJECT`. M80's `*TITLE` and `$TITLE` are written `SUBTTL` in TATARA. Appendix J.

`out of mapper memory.`

The memory mapper has no room left for TATARA's tables. This is not an internal error: the source needs more memory than the MSX has. Section 20.5.

`phase error - this label had a different value on pass 1.`
A label has a different address on the second pass, usually because a conditional gave a different answer on each pass. Section 15.2.

`relocation error - a segment-relative value is not allowed here.`
An address that TANREN has to finish, or an external name, is used where it cannot be: in one byte, multiplied, as the count of `REPT`, as the operand of `ORG` in another segment, or subtracted from a label in another segment. Section 8.6.

`RST takes 0, 8, 10h and so on to 38h.`
The operand of `RST` is not one of the eight restart addresses. Section 9.6.

`source line too long.`
A line of a source file is longer than 255 characters. Section 6.6.

`sources nested too deeply.`
Include files, macro expansions and repeat blocks inside each other are more than 16 deep, often because a macro uses itself without an end. Section 16.7.

`Tatara needs a memory mapper.`
The MSX has no memory mapper. Printed before the banner. Section 5.6.

`Tatara needs MSX-DOS2 or Nextor.`
The MSX is running MSX-DOS1. Printed before the banner. Section 5.6.

`the output file is the input file.`
The object file has the same name as the source. Section 5.2.

`the TATARA variable is too long.`
The value of `TATARA` is longer than 127 characters. A value set with **`set`** at the prompt or in a batch file cannot be that long, because MSX-DOS gives a command 127 characters; only a value set by a program can. Section 13.3.

`this label was defined on pass 1 and not on pass 2 - a conditional skipped it.`
A conditional assembled a label on the first pass and skipped it on the second, for example a label inside `IF1`, or an include file read twice. Section 15.2.

`this name already has a value.`
A name is defined twice: a label or `EQU` name defined again, a `DEFL` name that is also a label or `EQU` name, or a name both declared `EXTRN` and defined. In a macro, use `LOCAL` for its labels. Section 12.2.

`this segment was declared differently before.`
A segment is selected as transient in one place and not in another.

Section 10.7.

`this transient DSEG needs a GROUP first.`
A line in a transient data segment reserves space before the first `GROUP`. Section 10.7.

`too many different source files.`
TATARA remembers the names of 32 file openings, the source included, and a 33rd was asked for. The same file included twice counts twice. Appendix E.

`too many filenames.`
More than three filenames on the command line. Section 5.6.

`too many macro parameters.`
A macro has more than 16 parameters. Section 16.9.

`too many segments, groups or externals.`
The source has more named segments, groups or external names than an object file can hold (appendix E). Section 10.7.

`too many source files open at once.`
More than four files are open at once, through include files that include others, or a file that includes itself. Section 13.5.

`undefined symbol in an expression.`
A name in an expression is not defined anywhere, or, in `IF` or `REPT`, not yet defined at that line. A hexadecimal number that starts with a letter needs a 0 in front. Section 15.2.

`unknown option.`
A slash is followed by a letter that is not one of TATARA's options, or by nothing. Printed before the banner. Section 5.6.

D.3 TANREN's messages

`/D: would put the data on top of the code.`
The address given with `/D:` is inside the code. Section 24.3.

`a file list may not name another one.`
A word in a link file starts with `@`. Section 22.4.

`cannot create` *`NAME`*
The output file cannot be made, for example because its directory does not exist. Section 26.9.

`cannot open` *`NAME`*`.TRO`
No object file of that name was found in the places TANREN looks. Section 23.1.

`cannot open the file list` *`NAME`*`.LNK`
: There is no link file of that name, or `@` has no name after it. Section 22.4.

`cannot write the output file - the disk may be full.`
: The output file was made, but writing to it failed. TANREN stops at once and leaves the part it wrote; delete it before linking again. Section 27.2.

`not a Tatara object file.`
: The message begins with the name of the object file. The file does not start as an object file does. Section 27.1.

`one module was assembled /C and another was not.`
: All the modules must be assembled with `/C`, or all without it. Section 25.5.

`only one file list may be given.`
: The command line names more than one link file. Section 22.4.

`out of mapper memory.`
: The memory mapper has no room left for the program's image or for TANREN's tables. Section 26.7.

`segment` *`NAME`* `is declared differently.`
: The message begins with the name of the object file. The segment is a different kind in this file and in an earlier one. Section 27.4.

`Tanren needs a memory mapper.`
: The MSX has no memory mapper. Printed before the banner. Section 21.6.

`Tanren needs MSX-DOS2 or Nextor.`
: The MSX is running MSX-DOS1. Printed before the banner. Section 21.6.

`that public symbol is already defined.`
: Two modules define the same public name. The line before the message, `NAME is defined again in module MOD`, names it. Section 25.4.

`the linked image would run past FFFFh.`
: The program does not fit below FFFFh from the address it starts at. Section 24.6.

`the object file ends inside a record.`
: The message begins with the name of the object file. The file stops in the middle of its contents. Assemble the module again. Section 27.3.

`the output file` *`NAME`* `is also an input file.`
: The output file would overwrite one of the object files. Section 21.2.

`the symbols above were never defined.`
: The names listed under `Never defined:` are declared external by some module and defined by none. Section 25.3.

`the TANREN variable is too long.`
: `TANREN` is longer than 127 characters. Section 23.5.

`this object file was made by another version.`
: The message begins with the name of the object file. The file's format is not the one this TANREN reads. Assemble the module again with the TATARA of the same version. Section 27.3.

`too many external symbols in one module.`
: A module declares more than 512 external names. TATARA itself stops at 254, so an object file made by TATARA cannot reach this. Appendix E.

`too many segments or groups in one module.`
: The message begins with the name of the object file. The module has more than 16 segments or more than 16 groups, the two default segments included. Section 27.4.

`too many words on the command line or in the file list.`
: The words take more than 4096 characters in all, or one word is longer than 128 characters. Section 22.4.

`unknown option.`
: An option TANREN does not have, or `/D`, `/O` or `/P` without a colon. Printed before the banner. Section 21.3.

D.4 TANREN's warnings and notes

These are not errors: TANREN goes on, and the link succeeds.

`FILE``: bytes follow the END record.`
: Not an error: the object file has bytes after its last record. The summary line then ends `AND BYTES AFTER THEM.` instead of `ends at EOF.` `/Q` hides both. TATARA does not make such a file: assemble the module again. Section 27.3.

`Nothing to write: no module has any content.`
: Not an error: no module has a byte with content, so no file is made. Section 26.8.

`WARNING:` *`NAME`* `is entered at` *`XXXX`*`, not at` *`YYYY`*`.`
: The file was written without `/B`, so it will be entered at its first address, *XXXX*, but the address on the `END` line is *YYYY*. For a `.COM` program, put the code named by `END` first. Printed even with `/Q`. Section 26.5.

`WARNING: the command line is 127 characters, which is all MSX-DOS gives - anything past it was dropped in silence.`
: A third line follows: `Use @<file> if a module is missing.` The command line is exactly 127 characters long, so part of it may have been cut off. Put the link in a link file. Section 21.5.

APPENDIX E

Limits

E.1 TATARA 321
E.2 TANREN 322
E.3 The command line 323
E.4 What has no fixed limit 323

This appendix lists the fixed limits of TATARA and TANREN: how far each one goes, the message that going past it gives, and the section that describes it. Section E.4 says what has no fixed limit.

E.1 TATARA

What	Limit	Message	Section
A source line	255 characters	`source line too long.`	6.6
A symbol name	255 characters, all significant		7.2
A macro name	64 characters	`a macro name may be at most 64 characters.`	7.2
A segment or group name	16 characters	`a segment or group name may be at most 16 characters.`	7.2
Filenames on the command line	3	`too many filenames.`	5.6
Files open at once	4	`too many source files open at once.`	13.5
Include files, macro expansions and repeat blocks inside each other	16	`sources nested too deeply.`	16.7

What	Limit	Message	Section
Files opened in one assembly, the source included	32	`too many different source files.`	D.2
The name of a file at the start of a message	its first 31 characters are shown		20.1
Conditionals open at once	16	`conditionals nested too deeply.`	15.6
Macro parameters	16	`too many macro parameters.`	16.9
A line of a macro or repeat block, once expanded	255 characters	`macro line too long.`	16.2
External names	254	`too many segments, groups or externals.`	12.5
A title or subtitle	its first 28 characters are kept		18.2
The length of a page	10 to 255 lines	`a PAGE length must be 10 to 255.`	18.2
The `TATARA` variable	127 characters	`the TATARA variable is too long.`	13.3

A macro, segment or group name that is too long is refused, not cut short. A file opened twice counts twice towards the 32 files of one assembly. The name of a file at the start of a message is cut to its first 31 characters, so a long path can hide the name of the file itself; the `included from` lines and the `cannot open` message show names in full.

TATARA accepts more named segments and groups than TANREN does, so in practice the limits of section E.2 apply.

E.2 TANREN

What	Limit	Message	Section
Segments in one module	16, the default code and data segments included	`too many segments or groups in one module.`	27.4

What	Limit	Message	Section
Groups in one module	16	`too many segments or groups in one module.`	27.4
External names in one module	512	`too many external symbols in one module.`	D.3
The words of a link, command line and link file together	4096 characters	`too many words on the command line or in the file list.`	22.4
One word	128 characters	the same	22.4
Link files	1, which may not name another	`only one file list may be given.,a file list may not name another one.`	22.4
The `TANREN` variable	127 characters	`the TANREN variable is too long.`	23.5
The program	ends at FFFFh or below	`the linked image would run past FFFFh.`	24.6

A module can therefore have up to fourteen named segments, besides the default code and data segments. The limits are for each module, not for the program: a program that needs more can spread them over several modules. TATARA stops at 254 external names, so the limit of 512 is never reached by an object file that TATARA made.

E.3 The command line

MSX-DOS gives a program a command line of 127 characters, and drops the rest without a message. This is a limit of MSX-DOS, not of the tools, and it applies to both. TANREN warns when its command line is exactly 127 characters long, because part of it may be missing; a link file has no such limit (section 21.5).

E.4 What has no fixed limit

The number of symbols, the number and size of macros, and the size of the program are limited only by the memory mapper, where TATARA keeps its tables and TANREN builds the program. When the mapper is full, the tool stops with `out of mapper memory.` Appendix G describes how the tools use memory.

APPENDIX F

The `.tro` object file format

F.1 What an object file says 325
F.2 A worked example 326
F.3 File layout 329
F.4 The records 330
F.5 How TANREN reads an object file 332

TATARA writes, and TANREN reads, *object files* with the extension `.tro`. This appendix describes what such a file holds and how it is laid out, for anyone who wants to write a program that reads or writes one, or to understand what TANREN's `/R` prints. It describes the format as TATARA and TANREN version 1.2.0 use it: version 2 of the format. It first explains what an object file has to say, then goes through a real one byte by byte, and only then gives the reference.

F.1 What an object file says

Chapter 2 explained why an assembler cannot finish a program on its own: it does not know where the linker will put each piece, nor the addresses of names that other modules define. An object file therefore tells the linker four things about its module:

1. **Its pieces**: the segments it puts bytes into, how many bytes each one has, and the bytes themselves.
2. **The names it offers**: each public name, with its value as an offset in one of the segments, or as a plain number.
3. **The names it needs**: each external name.

4. **The holes**: every word in the bytes whose value depends on where a segment is placed, or on an external name.

A hole is not left empty. The word holds the part of its value that is already known, the *addend*, and a separate *fixup* says what to add to it. A fixup holds no number of its own: it only names what to add, and the number is always in the word. In `ld hl,count`, where `count` is at offset 0000h of the data segment, the word holds 0000h, and the fixup says "add the address of the data segment". In `ld hl,count+2`, the word holds 0002h, the offset of `count` plus 2, and the fixup is the same. A call to an external name holds 0000h, and its fixup says "add the value of that name". TANREN copies the bytes into place, and then adds what each fixup says.

Each of these is stored as a *record*: a type byte, the length of what follows, and the contents. A program that reads the file can skip a record it does not need by its length alone.

F.2 A worked example

This module has one of each of the things above:

```
; OBJ.AS - a module with one of each thing an object file holds.

                public  start,count
                extrn   putc

                cseg
start:          ld      hl,count        ; a word in the data segment
                ld      a,(hl)
                call    putc            ; an external
                ret

                dseg
count:          db      3

                dseg    scratch,transient
                group   one
buf1:           ds      4
                group   two
buf2:           ds      2

                end     start
```

`tatara obj.as obj.tro` makes a file of 169 bytes. Figure F.1 shows it as a row of records, and table F.1 goes through it byte by byte. Offsets and bytes are in hexadecimal, and the bytes of a string are shown as its text.

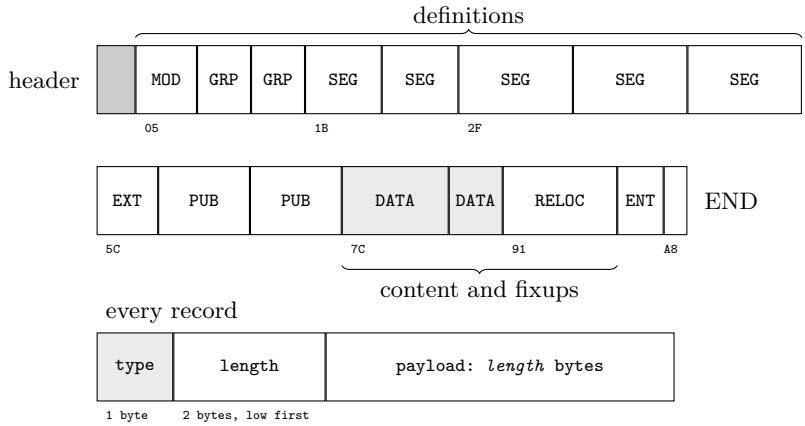


Figure F.1: `OBJ.TRO` as a row of records, and the layout of one record. The width of each box is its size in bytes.

Table F.1: `OBJ.TRO`, byte by byte.

Offset	Bytes	Meaning
00	`54 52 4F 1A 02`	the header: `TRO`, 1Ah, version 2
05	`01 05 00`	**MODNAME**, 5 bytes:
	`00 03 "obj"`	flags 00 (no `/C`); the name, `obj`
0D	`02 04 00 03 "one"`	**GRPDEF**: group 00 is `one`
14	`02 04 00 03 "two"`	**GRPDEF**: group 01 is `two`
1B	`03 07 00`	**SEGDEF**, 7 bytes: segment 00
	`00 FF 08 00 02 " C"`	code, no group, 8 bytes, the default code segment
25	`03 07 00`	**SEGDEF**: segment 01
	`01 FF 01 00 02 " D"`	data, no group, 1 byte, the default data segment
2F	`03 0C 00`	**SEGDEF**: segment 02
	`03 FF 00 00 07 "scratch"`	transient data, no group, 0 bytes: the segment itself
3E	`03 0C 00`	**SEGDEF**: segment 03
	`03 00 04 00 07 "scratch"`	transient data, group 00, 4 bytes
4D	`03 0C 00`	**SEGDEF**: segment 04
	`03 01 02 00 07 "scratch"`	transient data, group 01, 2 bytes
5C	`05 05 00 04 "putc"`	**EXTDEF**: external 0000 is `putc`
64	`04 09 00`	**PUBDEF**, 9 bytes:

Offset	Bytes	Meaning
	`01 00 00 05 "count"`	segment 01, offset 0000h: `count`
70	`04 09 00`	**PUBDEF**:
	`00 00 00 05 "start"`	segment 00, offset 0000h: `start`
7C	`06 0B 00`	**DATA**, 11 bytes:
	`00 00 00`	segment 00, offset 0000h
	`21 00 00 7E CD 00 00 C9`	the code: `ld hl,count`, `ld a,(hl)`, `call putc`, `ret`, with 0000h in both holes
8A	`06 04 00 01 00 00 03`	**DATA**: segment 01, offset 0000h, the byte 03
91	`07 0C 00`	**RELOC**, 12 bytes, two fixups:
	`00 00 01 00 01 00`	segment-relative: the word at offset 0001h of segment 00, plus the address of segment 01
	`01 00 05 00 00 00`	external: the word at offset 0005h of segment 00, plus the value of external 0000
A0	`08 03 00 00 00 00`	**ENTRY**: segment 00, offset 0000h
A6	`FF 00 00`	**END**

A few things are worth noticing:

- The default code and data segments have names, `" C"` and `" D"`: a space and a letter, which no source can write as the name of a segment. Every object file has both, even when one of them is empty.
- The transient segment `scratch` has a **SEGDEF** of its own, with no group and no bytes, and then one for each group, each with the size of that group.
- The name of the module is the name of the source as it was typed on the command line.

Linked with `PUTC.TRO`, which defines `putc` as a single `ret`, TANREN puts the code of `OBJ` at 0100h, the code of `PUTC` after it at 0108h, and the data segment at 0109h. It copies the bytes into place and applies the two fixups: 0000h + 0109h for `ld hl,count`, and 0000h + 0108h for `call putc`. The program, `OBJ.COM`, is these 10 bytes:

```
21 09 01 7E CD 08 01 C9 C9 03
```

The transient segment is at 010Ah, with both groups starting there. It holds no bytes, so it is not part of the file.

F.3 File layout

A file holds one module. It starts with a header of five bytes:

Offset	Bytes	Meaning
0	`54 52 4F`	the letters `TRO`
3	`1A`	the end-of-file mark of MSX-DOS: **`type obj.tro`** prints `TRO` and stops
4	`02`	the version of the format

The records follow, up to an **END** record. Each one is a type byte, a word with the length of the payload, and the payload. These rules apply throughout:

- A *word* is two bytes, the low byte first, as the Z80 stores them.
- A *string* is a length byte, from 1 to 255, followed by that many characters. Names are stored exactly as they were written; the flags of **MODNAME** say whether upper and lower case were told apart.
- Segments, groups and external names are numbered in the order of their records, from 0. A segment or group number is a byte, and FFh means none: an absolute value for a segment, no group for a group. An external number is a word.
- A record can use only the numbers of segments, groups and external names whose records came before it.
- Anything after the **END** record is ignored. TANREN notes it, with `bytes follow the END record.` (section 27.3).

TATARA writes the records in this order: **MODNAME**, the **GRPDEF**s, the **SEGDEF**s, the **EXTDEF**s, the **PUBDEF**s, the **DATA** records, **RELOC**, **ENTRY** if there is an `END` address, and **END**.

F.4 The records

Each record is described below with the layout of its payload and the line that TANREN's /R prints for it (section 21.3). In the /R lines, values are in hexadecimal.

01h MODNAME: the module

Field	Size	Meaning
flags	byte	bit 0 set: assembled with /C; the other bits are 0
name	string	the name of the module

The first record of the file, and the only one of its type. TANREN compares module names without regard to case. /R prints `MODNAME  flags 00 obj`.

02h GRPDEF: a group

The payload is the group's name, a string. The group gets the next group number. Groups of the same name are the same group, in this module and in every other (section 10.6). /R prints `GRPDEF    one`.

03h SEGDEF: a segment

Field	Size	Meaning
flags	byte	bit 0 set: data, clear: code; bit 1 set: transient
group	byte	the group number, or FFh for none
size	word	how many bytes this module puts in the segment
name	string	the name, `" C"` or `" D"` for the default segments

The segment gets the next segment number. TANREN joins the segments of the same name and group from every module, in the order of the modules, and lays the groups of a transient segment over one another. A segment that is code in one module and data in another, or transient in one and not in another, is an error. /R prints `SEGDEF   flags 03 group 00 size 0004 scratch`.

04h PUBDEF: public names

Field	Size	Meaning
segment	byte	the segment of the value, or FFh for a plain number
offset	word	the value: an offset in that segment, or the number
name	string	the name

The three fields can be repeated to fill the payload; TATARA writes one name to a record. `/R` prints `PUBDEF  seg 01 offset 0000 count`.

05h EXTDEF: external names

The payload is one or more strings, each an external name, which gets the next external number. `/R` prints `EXTDEF  putc`.

06h DATA: bytes

Field	Size	Meaning
segment	byte	the segment, or FFh for the absolute segment
offset	word	where the bytes start in the segment, or the address
bytes	the rest	the bytes, with the addends in the holes

The record's length counts the three bytes before the bytes, so 11 for 8 bytes. A segment can have any number of **DATA** records, in any order. `/R` prints the number of bytes itself, and the bytes: `DATA  seg 00 offset 0000 len 0008 21 00 00 7E CD 00 00 C9`.

07h RELOC: fixups

Field	Size	Meaning
kind	byte	00h: add a segment's address; 01h: add an external's value
segment	byte	the segment that holds the word
offset	word	where the word is in that segment
target	word	kind 00h: the segment number; kind 01h: the external number

The six bytes are repeated for each fixup. Every fixup is for a whole word: there is no fixup for a single byte, which is why a one-byte operand must be absolute (section 8.6). `/R` prints the count: `RELOC  2 fixups`.

08h ENTRY: the start address

A segment byte and an offset word: the address on the `END` line. At most one in a module; TANREN uses the first it finds among the modules (section 26.5). `/R` prints `ENTRY  seg 00 offset 0000`.

7Fh COMMENT

Any bytes, which a reader skips. TATARA does not write this record.

FFh END

An empty payload. The last record of the module. `/R` prints `END`.

F.5 How TANREN reads an object file

TANREN reads every object file twice, from start to end (section 27.2). Figure F.2 shows the two passes over `OBJ.TRO` and `PUTC.TRO`, the example of section F.2.

1. **The first pass** reads the definitions: the names of the modules, the groups and segments with their sizes, the public names and the external names, and

skips **DATA** and **RELOC** by their length. When it has read every file, it places the segments (chapter 24), works out the value of every public name, and checks that every external name is defined.

2. **The second pass** copies the bytes of each **DATA** record into place, and adds to each word what its fixup says. Then TANREN writes the output file.

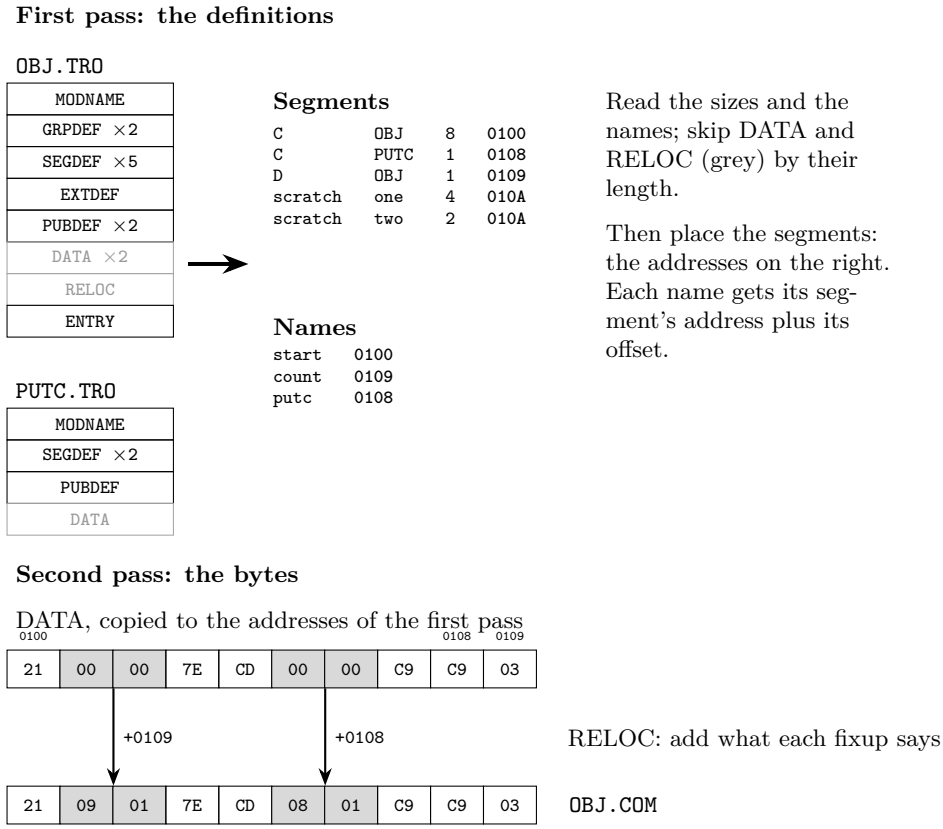


Figure F.2: What TANREN does with `OBJ.TRO` and `PUTC.TRO`. Above, the first pass builds the tables of segments and names; below, the second pass copies the bytes and fills the holes (shaded).

APPENDIX G

Memory

G.1 The memory map 335
G.2 What goes into the mapper 336
G.3 How much memory the tools need to build a program 336
G.4 When the mapper is full 338

TATARA and TANREN keep their tables, and TANREN keeps the program it builds, in the MSX's memory mapper, not in the 64 KB that the Z80 can see. This appendix describes how the tools use memory, how much they need to build a program, and what to do when there is not enough. It is about the memory the tools need while they work, not about the memory the finished program needs when it runs.

G.1 The memory map

A *memory mapper* is RAM divided into segments of 16 KB. The Z80 sees four of them at a time, one in each quarter of its 64 KB, which are called *pages*. Figure G.1 shows how the four pages are used while TATARA or TANREN runs:

- **0000h–3FFFh and 4000h–7FFFh**: MSX-DOS uses the bytes below 0100h; from 0100h, the tool itself, its code and the data it needs all the time.
- **8000h–BFFFh**: the window. The tool brings one segment of the mapper at a time into this page, to read or write what it keeps there, and gives the page back to MSX-DOS before each call to it.
- **C000h–FFFFh**: MSX-DOS and the stack.

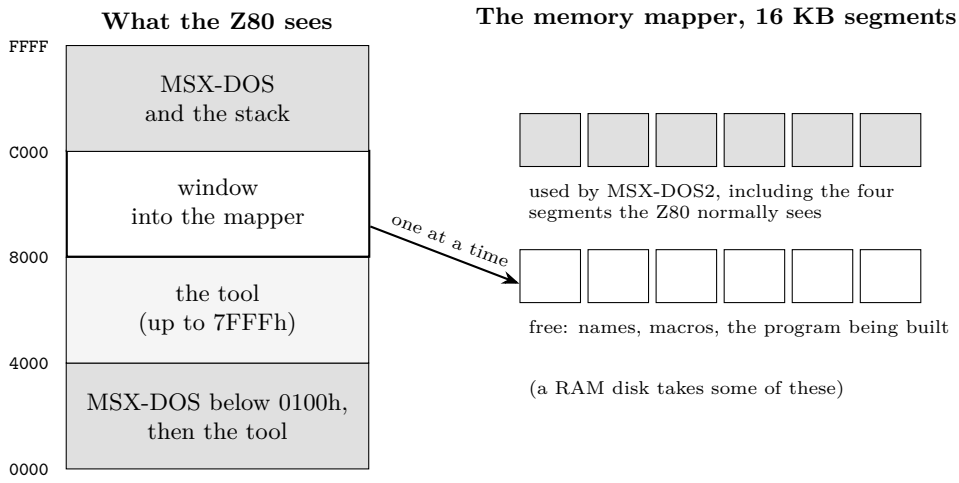


Figure G.1: The Z80's 64 KB while TATARA or TANREN runs, and the mapper segments behind the window.

G.2 What goes into the mapper

The segments that MSX-DOS2 has not given to anything else are *free*, and the tools ask MSX-DOS2 for them as they need them:

- **TATARA** keeps there the names and their values, the bodies of macros, and the records it will write into the object file.
- **TANREN** keeps there the segments and names of every module, and the program it is building. The program takes one segment for each 16 KB of addresses it covers: a program from 0100h to A0FFh covers three. Addresses that nothing is written to take no segment.

So the size of a program is limited by the free memory of the mapper, not by the memory that MSX-DOS leaves for programs below C000h.

G.3 How much memory the tools need to build a program

What counts is the free memory, not the total. MSX-DOS2 takes part of the mapper for itself: 112 KB on an MSX2 or MSX2+ and 176 KB on an MSX turbo R, in the tests below, without a RAM disk. A RAM disk takes more.

Table G.1 shows what fitted in the free memory of several machines. Each source defined names of 16 characters with `EQU`, and the program was 40 KB of data, linked as a `.COM` program.

MSX	Total	MSX-DOS2	Free	2000 names	8000 names	40 KB program
MSX2	128 KB	112 KB	16 KB	no (495)	no (495)	no
MSX2	192 KB	112 KB	80 KB	yes	no (2479)	yes
turbo R	256 KB	176 KB	80 KB	yes	no (2479)	yes
turbo R	384 KB	176 KB	208 KB	yes	no (6447)	yes
turbo R	512 KB	176 KB	336 KB	yes	yes	yes
turbo R	1024 KB	176 KB	848 KB	yes	yes	yes

Table G.1: What fits in the free memory of the mapper. A number in brackets is how many names fitted.

These rules of thumb follow from it:

- Each 16 KB of free memory holds about 495 names of 16 characters, so a name takes about 33 bytes. Shorter names take less.
- TANREN needs one free segment for each 16 KB of addresses the program covers, as well as room for its tables.
- With 16 KB free, the least that MSX-DOS2 leaves on an MSX with 128 KB, the tools run, but can build only small programs.
- With 80 KB free, the tools build themselves (chapter 35): a program of 27 modules and some 30 KB. An MSX2 with 192 KB, or a turbo R with 256 KB, has that much.
- With 336 KB free or more, everything in the table fits. A mapper of 512 KB is comfortable for any program that fits in an MSX.

Most cartridges that give an MSX2 the MSX-DOS2 kernel bring a mapper of their own: ASCII's MSX-DOS2 cartridge has 128 KB, the SD-512 and the MegaFlashRom SCC+ SD 512 KB, and the Carnivore2 1 MB. A cartridge with only a mapper, such as the MEGA-MAPPER with up to 4 MB or the 16 MB memory cartridge V3, adds memory to any MSX. Appendix H lists these cartridges and where to get them.

How much memory an MSX has

The startup screen of an MSX shows how much memory it has, but some models show only the memory of the mapper inside the computer, not of one in a cartridge. `MAPPER.COM`, from `https://github.com/javilm/msx-mapper`, shows every mapper that MSX-DOS2 found, with how much of each MSX-DOS2 uses and how much is free:

MSX-DOS

```
A:\>mapper
Total 256Kb (176Kb sys, 0Kb user, 80Kb free).
Memory mappers detected by MSX-DOS2:
[PRIMARY] Expanded slot 3-0: 176Kb sys, 0Kb user, 80Kb free.
```

The `free` figure is the one that the tools can use.

G.4 When the mapper is full

When there is no free segment left, the tool stops with `out of mapper memory.` To get more room:

- **Remove the RAM disk**, if there is one: **`ramdisk 0`** gives its memory back to MSX-DOS2. Programs that stay in memory take mapper segments too.
- **Split the program** into more modules (chapter 28). TATARA needs room for the names of one module at a time.
- **Add memory**: a mapper cartridge, as above.

APPENDIX H

Where to get the hardware

H.1 What the tools need 339
H.2 The cartridges. 340
H.3 The makers and where to buy 340
H.4 Availability. 341

This appendix is for readers who want to run TATARA and TANREN on a real MSX and need to know what to add to it. Several small makers build cartridges for the MSX today, and some of them give it everything the tools need.

H.1 What the tools need

Besides an MSX2, MSX2+ or MSX turbo R, the tools need three things:

- **MSX-DOS2**, or *Nextor*, which runs the programs written for MSX-DOS2. MSX-DOS2 gives the tools the memory of the mapper (appendix G).
- **A memory mapper** with enough free memory. Appendix G recommends 512 KB (section G.3).
- **Somewhere to keep the files**. A floppy disk is enough for small programs, but a larger program, such as the tools themselves (chapter 35), is much easier to work with on an SD or CompactFlash card or a hard disk.

Most MSX2 and MSX2+ computers have a mapper of only 64 or 128 KB, and only the MSX turbo R has the MSX-DOS2 kernel built in. A single cartridge can supply all three.

H.2 The cartridges

Table H.1 lists cartridges that supply them, and what each one gives an MSX.

Table H.1: Cartridges that give an MSX what the tools need.

Cartridge	Mapper	Also has	Made by
SD-512	512 KB	Nextor; an SD and a microSD card slot	8bits4ever
MegaFlashRom SCC+ SD	512 KB, optional	Nextor; two microSD card slots; 8 MB of flash memory for ROMs; SCC+ sound	MSX Cartridge Shop
Carnivore2	1 MB	Nextor; a CompactFlash card slot; 8 MB of flash memory for ROMs; SCC and FM-PAC sound	RBSC
MEGA-MAPPER	up to 4 MB	nothing else: a mapper only	8bits4ever
16 MB memory cartridge V3	16 MB (4×4 MB)	nothing else: a mapper only	Classic PC Club
MSX-DOS2 cartridge	128 KB	MSX-DOS2	ASCII

The SD-512, the MegaFlashRom SCC+ SD (with its mapper module) and the Carnivore2 each give an MSX2 all three things on their own. The MEGA-MAPPER and the 16 MB cartridge add only memory, for an MSX that already has MSX-DOS2 and somewhere to keep the files, or that gets them from another cartridge.

The 16 MB cartridge has a slot expander inside it, and shows MSX-DOS2 four mappers of 4 MB, each in a slot of its own. MSX-DOS2 uses all of them, so the tools see 16 MB, less what MSX-DOS2 takes for itself.

ASCII's MSX-DOS2 cartridge is no longer made, but it is still found second-hand. Its 128 KB are enough to run the tools only for small programs (table G.1).

H.3 The makers and where to buy

8bits4ever makes the SD-512 and the MEGA-MAPPER, and sells the Carnivore2:
`https://www.8bits4ever.net/product-page/sd-512`

MSX Cartridge Shop makes the MegaFlashRom SCC+ SD:
`https://www.msxcartridgeshop.com`

RBSC (Russian Bear Service Crew) designs the Carnivore2 and its newer version, the **Carnivore2+**, which is another option:
`https://rbsc.su/en/completed_projects/`
`https://rbsc.su/en/projects_in_development/`

Classic PC Club (クラシックPC研究会) makes the 16 MB memory cartridge V3, made by a team of Kobayashi, Hanashima, Inoue, Suzuki and Azukino. It has been tested on the Sony HB-F1XDJ and HB-F1XV and on the Panasonic FS-A1, FS-A1ST and FS-A1GT:
`https://classicpc.org`

RBSC does not sell the Carnivore2 itself, but through resellers in several countries:

- 8bits4ever (Spain): `https://www.8bits4ever.net/product-page/carnivore2`
- HobbyRetro (Spain): `https://hobbyretro.com/en/retro/carnivore-2`
- Retro Game Restore (Taiwan): `https://retrogamerestore.com/store/carnivore2/`
- The Retro Hacker Store (Brazil): `https://www.theretrohacker.com/product/rbsc-carnivore2-msx-flashrom-scc-fmpac-ide-cf-ram/`
- Denjhang's Retro Hardware (China): `https://denjhang.com/products/carnivore2-msx-multifunctional-cartridge`
- Maxiol (Russia), which also sells the Carnivore2+: `https://radioshop.maxiol.com/product/carnivore-2-for-msx`

H.4 Availability

These cartridges are made by small makers, in small numbers, and they come and go from the shops as each batch is made and sold. A cartridge that is not listed today, as the MEGA-MAPPER was when this manual was written, may be back with the next batch, and the web addresses may change. The makers' own sites are the best place to ask.

Without any of them, an emulator such as openMSX can emulate an MSX with MSX-DOS2, a mapper of any size and a hard disk, and the tools run there as they do on a real MSX.

APPENDIX I

Environment variables

I.1 Setting a variable 343
I.2 PATH 344
I.3 TATARA 344
I.4 TANREN 345
I.5 The rules both share 346

The tools depend on three environment variables. `PATH` lets MSX-DOS2 find them, `TATARA` tells TATARA where to look for include files, and `TANREN` tells TANREN where to look for object files. They are introduced where they are first needed, in chapters 3, 13 and 23. This appendix gathers the rules in one place. The tools read no other environment variable.

I.1 Setting a variable

An environment variable is set with the `SET` command of MSX-DOS2, at the prompt or in a batch file:

MSX-DOS

```
A:\>set tatara="a:\tatara\include"
```

- **At the prompt**, write a value that contains \ between quotation marks, as above. Without them, MSX-DOS2 answers `*** Invalid environment string` and does not set the variable.
- **In a batch file**, the quotation marks are not needed. If they are written, `SET` removes them, and the value is the same.

- **The value a variable already has** is written `%name%`. The first line of the `AUTOEXEC.BAT` example below uses `%path%` to add a directory to the end of `PATH` instead of replacing it.
- **To list** every variable and its value, type **`set`** on its own.
- **To remove** a variable, set it to nothing: **`set tanren=`**.
- **The name** can be written in upper or lower case: MSX-DOS2 keeps it in upper case, so **`set Tatara=inc`** sets `TATARA`.

A variable lasts until the MSX is reset or switched off. To set them every time the MSX starts, put the `SET` commands in `AUTOEXEC.BAT`, in the root directory of the boot drive (section 3.4). With all three:

```
set path=%path%;a:\tatara\bin
set tatara=a:\tatara\include;a:\project\inc
set tanren=a:\project\lib
```

A command line, a `SET` command included, is at most 127 characters long, after each `%name%` has been replaced by its value. MSX-DOS2 refuses a longer one with `*** Command too long` and leaves the variable as it was.

I.2 PATH

`PATH` belongs to MSX-DOS2, not to the tools. When a command is typed, MSX-DOS2 looks for a program of that name in the current directory and then in each directory of `PATH`, so naming the directory of `TATARA.COM` and `TANREN.COM` in it lets them be run from any directory. The tools themselves do not read `PATH`: it plays no part in finding include files or object files.

I.3 TATARA

TATARA looks for a file named by `INCLUDE` in three places (section 13.2, figure 13.1):

1. the directory of the file that includes it;
2. the current directory;
3. each directory named in `TATARA`, from left to right.

The variable is used only when the file is in neither of the first two places, and never for an absolute name, one that starts with a drive or with `\` (section 13.4). TATARA reads it the first time it needs it, and keeps the value until the end of the assembly.

A relative name with a directory in it, such as `sub\mid.inc`, is joined to each directory of the variable in the same way as a plain name. Figure I.1 shows how a value and a name become the names that TATARA tries.

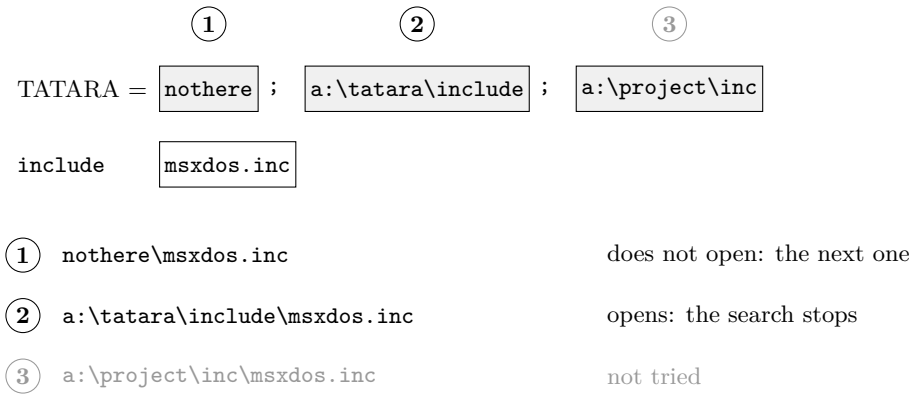


Figure I.1: How the value of `TATARA` and the name in an `INCLUDE` line become the names that are tried, in order.

I.4 TANREN

TANREN looks for an object file in three places too (section 23.1, figure 23.1):

1. the directory of the link file that names it, if it was named in one;
2. the current directory;
3. each directory named in `TANREN`, from left to right.

As with `TATARA`, the variable is used only when the file is in neither of the first two places, never for an absolute name, and is read once for the whole link.

Without `/O:`, the output file goes into the directory where the first object file was found. If that object file came from a directory of `TANREN`, the output file goes there too (section 23.4); name the output file with `/O:` to put it somewhere else.

I.5 The rules both share

`TATARA` and `TANREN` are read in the same way. Each directory of the value, followed by `\` and the name, is tried in turn, and the first one that opens is used. Table I.1 gives the details.

Table I.1: How `TATARA` and `TANREN` are read.

What	What happens
Separator	`;` between directories, as in `PATH`.
Order	From left to right. The first directory where the file opens is used, and the rest are not tried.
A `\` at the end	Not needed, and does no harm: `a:\inc` and `a:\inc\` are the same.
A drive alone	`b:` means the current directory of drive B:, as it does in MSX-DOS2.
A relative directory	Relative to the current directory, not to the source file or the link file: `inc` is `INC` under the directory the tool was started in.
A root-relative directory	`\project\inc` is on the current drive.
A directory that does not exist	Passed over without a message; the next one is tried.
An empty entry	`;;` or a `;` at the start is the current directory, which has already been tried, so it does no harm.
Spaces	Part of the directory name. In `a:\inc; a:\lib` the second directory starts with a space, so it does not exist and is passed over. Do not put a space after the `;`.
Upper and lower case	The same, as everywhere in MSX-DOS.
A long directory	The directory, `\`, and the name together must fit in 63 characters. A directory that is too long for the name is passed over without a message, even if the file is there.
A long value	At most 127 characters. A longer value stops the tool with `the TATARA variable is too long.` or `the TANREN variable is too long.` rather than letting it search a shortened list. `SET` cannot make a value that long, because its own command line is limited to 127 characters, so the message appears only if another program set the variable.
No variable	Only the first two places are searched.

When a file is not found anywhere, the message names the file as it was written, not the names that were tried:

MSX-DOS

```
A:\>tatara which.as
...
ERROR: cannot open ENVT.INC
    included from WHICH.AS(2)
```

`SET` on its own shows what the variable holds, which is the first thing to check when a file that exists is not found.

On a Japanese MSX, `\` appears as `¥`. It is the same character.

APPENDIX J

Source compatibility with M80

J.1 What carries over 349
J.2 M80 directives that TATARA does not accept 350
J.3 Other forms 351
J.4 What TATARA adds 352
J.5 Other differences 353
J.6 Converting a source 353

This appendix is for readers who have sources written for M80 and want to assemble them with TATARA. It compares TATARA with MSX-M80, the M80 in ASCII's MSX-DOS2 TOOLS, whose listings call it `MSX.M-80 2.00`.

Most Z80 sources written for M80 assemble with TATARA unchanged. A few directives and forms need changing, and are listed below with what to write instead. Object files are not interchangeable: TATARA writes object files of its own (appendix F), not M80's `.REL` files.

J.1 What carries over

These work in TATARA as they do in M80:

- The statement format: a label, the operation, the operands and a comment, with the label in column 1 or followed by a colon, and `name::` for a label that is also `PUBLIC` (chapter 6).
- Numbers with the suffixes `B`, `D`, `O`, `Q` and `H`, character constants of up to two characters (`''` is 0), strings in either kind of quotation mark, and `$` for the location counter. A number larger than 16 bits keeps its low 16 bits, as in M80.

- The operators and their levels (appendix C).
- `ASEG`, `CSEG`, `DSEG` and `ORG` (chapter 10).
- `DB`, `DW`, `DS` and `DC`, and `EQU` and `DEFL`.
- `PUBLIC`, `EXTRN` and `END`.
- The conditionals, from `IF` to `IFDIF`, with `ELSE` and `ENDIF` (chapter 15).
- `MACRO`, `REPT`, `IRP`, `IRPC`, `LOCAL`, `EXITM` and `ENDM`, and the special forms of macros: `&`, `;;`, `!`, angle brackets, and `%` for an argument passed by value (chapters 16 and 17).
- `TITLE`, `SUBTTL`, `PAGE` and the listing controls, from `.LIST` to `.TFCOND` (chapter 18).

M80's other names for directives are accepted too: `COND` and `IFT` for `IF`, `IFF` for `IFE`, `DEFB` and `DEFM` for `DB`, `DEFS` for `DS`, `DEFW` for `DW`, `ENTRY` for `PUBLIC`, `EXT` for `EXTRN`, and `*EJECT` and `$EJECT`, in column 1, for `PAGE` (section A.4).

J.2 M80 directives that TATARA does not accept

The directives in table J.1 stop TATARA with `not a directive, a macro or an instruction.`, or, for the two control lines, with `only *EJECT and its dollar spelling are control lines.`

Table J.1: M80 directives that TATARA does not accept, and what to write instead.

M80	In TATARA
`SET`	`DEFL`, which does the same (section 12.4).
`GLOBAL`	`PUBLIC`.
`EXTERNAL`	`EXTRN`.
`ENDC`	`ENDIF`.
`MACLIB`, `$INCLUDE`, `*INCLUDE`	`INCLUDE`.
`*TITLE`, `$TITLE`	`SUBTTL`, with the text without brackets or quotation marks.
`.Z80`	Delete it. TATARA always assembles Z80 instructions.
`.8080`	TATARA has no 8080 mode. The source has to be rewritten with Z80 mnemonics.

M80	In TATARA
`.RADIX`	Write each number with its suffix: `0FFh`, not `0FF` after `.radix 16`.
`COMMON`	Define the variables in one module and share them with `PUBLIC` and `EXTRN` (section 12.5); for blocks that take turns in the same memory, use a transient segment (chapter 29).
`NAME`	Delete it. The name of a module is the name of its source file (section F.1).
`.COMMENT`	Start each line of the comment with `;`, or put the block between `if 0` and `ENDIF`.
`.PRINTX`	No equivalent. Delete it.
`.REQUEST`	Name the file on TANREN's command line or in a link file (chapter 21).
`.PHASE`, `.DEPHASE`	No equivalent.
`.CREF`, `.XCREF`	Delete them. TATARA makes no cross-reference file.

J.3 Other forms

Table J.2 compares the other parts of a source that differ, or that a reader who knows M80 may wonder about.

Table J.2: Other forms in M80 sources, and how TATARA reads them.

M80 form	In TATARA
`foo::`	The same: a label that is also `PUBLIC`.
`far##`	Not accepted: `undefined symbol in an expression.` Declare the name with `EXTRN` and write it without `##`.
`X'41'`	Not accepted: `undefined symbol in an expression.` Write `41h`.
`NUL`	Not accepted. Test an argument with `IFB` or `IFNB` (section 16.6).
`TYPE`	Not accepted, and has no equivalent. `IFDEF` tells whether a name is defined.
`%lb` in a macro argument	The same: the value of `lb` is passed, in decimal.
`a!;b` and `<a;b>`	The same: both pass `a;b`, with the semicolon.
`;;` in a macro	The same: the comment is not kept in the expansion.

M80 form	In TATARA
An expression alone on a line	Not accepted. M80 takes `41h` on its own as `db 41h`; in TATARA, write the `DB`.
8080 mnemonics	Not accepted: `mvi a,1` is `ld a,1`.
A word in column 1 that starts with `*` or `$`	A control line, as in M80. Only `*EJECT` and `$EJECT` are accepted; any other stops TATARA, so `$loop:` cannot be a label.
A label with other characters	Refused: `a name may hold only letters, digits, ? @ . _ and the dollar sign.`
The length of a name	M80 uses only the first six characters of a name; TATARA uses all of them, up to 255 (section 7.2). Two names that M80 took as one, such as `buffer1` and `buffer2`, are two names in TATARA.
Upper and lower case	The same by default: they do not matter in a name. `/C` makes them matter (section 7.3).
The length of a line	M80 reads 132 characters; TATARA reads 255.
A label named `IXH`, `IXL`, `IYH` or `IYL`	Legal in M80, refused in TATARA, where the four are registers: `IXH, IXL, IYH and IYL are registers, not names.`

J.4 What TATARA adds

TATARA does more than MSX-M80 in these ways:

- **The undocumented instructions**: `SLL`, the index halves `IXH` to `IYL`, `IN F,(C)` and `OUT (C),0` (section 9.5). MSX-M80 has none of them.
- **Long symbol names**, all of whose characters count (section 7.2 and chapter 31).
- **Named segments** and `GROUP`, and **transient segments** for variables that share memory (section 10.5 and chapter 29).
- **A wider search for include files**: the directory of the including file, the current directory, then each directory of `TATARA` (chapter 13 and appendix I). MSX-M80 looks in one place only: the directory named by the environment variable `INCLUDE` if it is set, which MSX-C also uses, and the current directory if it is not.
- **MSX-DOS2 directories** in every filename (chapter 34).

- **Large programs**: the tables are kept in the memory mapper, not in the 64 KB the Z80 sees (appendix G).
- **A symbol table** with the segment and kind of each name, printed with `/S` (chapter 19).
- **Messages that say what is wrong**, with the file and line, and the chain of include files and macros that led to it (chapter 20).

J.5 Other differences

The command line. MSX-M80 takes a command such as **`m80 =hello/l`**, with the output files before the `=`. TATARA takes the source first: **`tatara /l hello.as`** (chapter 5).

The object file. TATARA writes a `.TRO` file, which TANREN links (appendix F). It cannot write a `.REL` file, and TANREN cannot read one, so every module of a program is assembled with TATARA.

The listing. The layout is close to M80's, with the differences noted in chapter 18.

Messages. M80 marks a line with a letter, such as U or V, and goes on to the end of the source. TATARA prints a sentence and stops at the first error (section 20.2, appendix D).

J.6 Converting a source

To assemble a source written for MSX-M80 with TATARA:

1. Delete `.Z80`, `NAME`, `.PRINTX`, `.CREF` and `.XCREF`.
2. Change `SET` to `DEFL`, `GLOBAL` to `PUBLIC`, `EXTERNAL` to `EXTRN`, `ENDC` to `ENDIF`, `MACLIB`, `$INCLUDE` and `*INCLUDE` to `INCLUDE`, and `$TITLE` or `*TITLE` to `SUBTTL`.
3. Replace `name##` with `EXTRN` declarations, `X'nn'` with `nnH`, and numbers written after `.RADIX` with numbers that carry their suffix.
4. Rewrite `.COMMENT` blocks, `COMMON` blocks and any expression alone on a line, as in tables J.1 and J.2.

5. Set `TATARA` to the directories of the include files, if they are not beside the source or in the current directory. TATARA does not read `INCLUDE`.

6. Rename any label called `IXH`, `IXL`, `IYH` or `IYL`.

7. Look for names that differ only after their sixth character: M80 took them as one name, TATARA does not.

8. Assemble. TATARA stops at the first line it cannot take, and the message names the file and line, so the rest can be found one at a time.

9. Assemble every module of the program with TATARA, and link them with TANREN.

APPENDIX K

The include files

K.1 MSXDOS.INC 356
K.2 ERRORS.INC 356
K.3 BIOS.INC 357
K.4 WORKAREA.INC 358
K.5 SUBROM.INC 358
K.6 HOOKS.INC 358
K.7 PORTS.INC 358
K.8 ASCII.INC 359
K.9 EXTBIO.INC 359
K.10 Further reading 359

The nine include files in `A:\TATARA\INCLUDE` hold 903 names and one macro. They save writing addresses and numbers by hand; they do not explain them. This appendix says what each file holds and where the names in it are described. Chapter 14 shows how to use them.

The names were taken from the MSX-Datapack, and are spelt as it spells them, except where a file says otherwise. Each name has a comment of one line in its file, and each file starts with a comment that explains how to use what is in it. What a routine needs in its registers, and what it gives back, is in the books listed in section K.10.

Table K.1 lists the files in the order of this appendix, the order in which a program is most likely to need them.

A program for MSX-DOS needs `MSXDOS.INC`, and usually `ERRORS.INC`. A program that runs from a cartridge, or is loaded from BASIC, needs `BIOS.INC` and `WORKAREA.INC` instead, and `SUBROM.INC` for what the MSX2 added. The rest are

File	What it holds	Names	Range
MSXDOS.INC	MSX-DOS function numbers, and the system macro	93	00h–70h
ERRORS.INC	MSX-DOS error codes	76	0FFh–81h
BIOS.INC	MAIN ROM entry points	92	0000h–017Dh
WORKAREA.INC	the system work area	311	0F323h–0FFFFh
SUBROM.INC	SUB ROM entry points	36	0089h–01F9h
HOOKS.INC	the hooks	117	0FD9Ah–0FFD4h
PORTS.INC	the I/O ports	56	07Ch–0FCh
ASCII.INC	the control codes	37	0–127
EXTBIO.INC	the extended BIOS, and the FM BIOS	85	

Table K.1: The include files, in the order of this appendix.

for programs that work with the hardware directly.

Figure K.1 shows where the addresses that the files name are in the Z80's 64 KB.

K.1 MSXDOS.INC

The MSX-DOS and MSX-DOS2 function numbers, from _TERM0 (00h) to _REDIR (70h), with BDOS, the entry at 0005h, and the system macro, which loads C and calls BDOS (section 14.3). The names are the official ones, those of the MSX-DOS2 sources, with their leading underscore. Numbers below 31h are in MSX-DOS1 too; 31h and above are MSX-DOS2 only.

Read about them in the MSX-DOS2 function specification, and in the MSX-Datapack, volume 3.

K.2 ERRORS.INC

The error codes of MSX-DOS and MSX-DOS2, from .NCOMP (0FFh) down to .OVDEST (81h), in four groups: disk errors, the errors the functions return, the errors that end a program, and the errors of the command interpreter. The names start with a dot, as the manual prints them. The file runs downwards, because the codes were given out counting down from 0FFh.

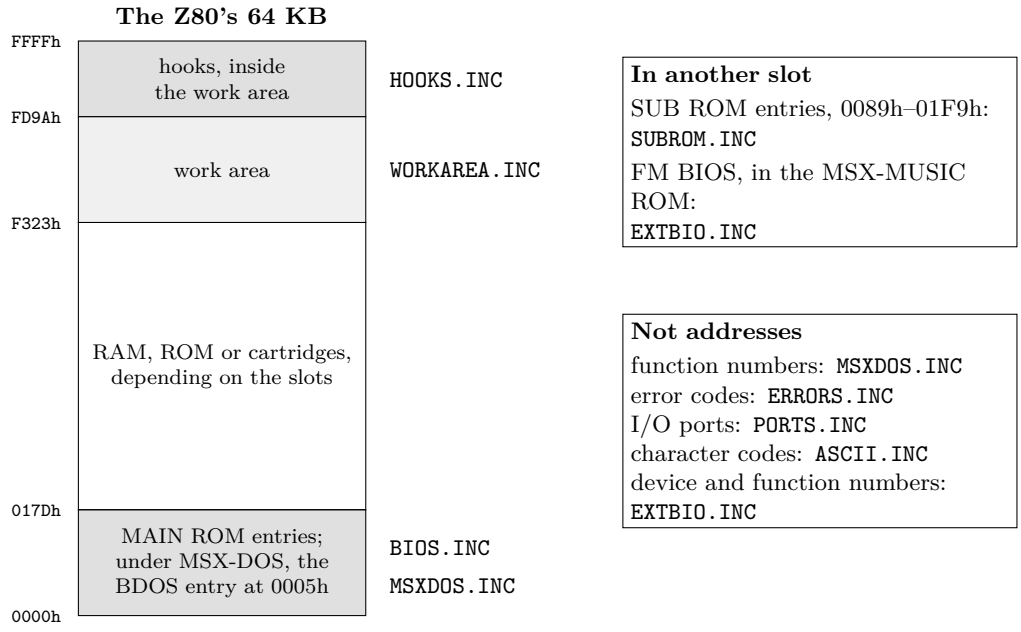


Figure K.1: Where the addresses named by the include files are.

Read about them in the MSX-DOS2 function specification, and in the MSX-Datapack, volume 3.

K.3 BIOS.INC

The entry points of the MAIN ROM, from `CHKRAM` (0000h) to `WRRES` (017Dh), in address order. The entries added for the MSX2 and the MSX2+ are marked in the file. The entries that the MSX turbo R added, for changing the CPU and for the PCM, are not in it.

Under MSX-DOS, only five of these can be called directly; the others need an inter-slot call (section 14.4).

Read about them in the MSX2 Technical Handbook and the MSX-Datapack, volume 2.

K.4 WORKAREA.INC

The system work area, the variables the BIOS and BASIC keep in RAM, from `DISKVE` (0F323h) to `EXPSLT` (0FFFFh). The comment of each name starts with its length in bytes, and the area is not one run of bytes: the gaps between the groups belong to the system and are not published.

Read about them in the MSX2 Technical Handbook and the MSX-Datapack, volume 2.

K.5 SUBROM.INC

The entry points of the SUB ROM, which the MSX2 added, from `SUBROM_GRPPRT` (0089h) to `SUBROM_WRTCLK` (01F9h). The SUB ROM is in another slot, so none of them can be called directly: they are reached through `EXTROM` or `SUBROM`, in `BIOS.INC`. Every name starts with `SUBROM_`, because the SUB ROM uses some of the MAIN ROM's names for other routines at other addresses.

Read about them in the MSX2 Technical Handbook and the MSX-Datapack, volume 2.

K.6 HOOKS.INC

The hooks, the five-byte entries in RAM that the system calls at fixed moments, from `H.KEYI` (0FD9Ah) to `ENAINT` (0FFD4h). They are inside the work area, and `EXTBIO`, the entry of the extended BIOS, is among them. Two hooks have two names each, an older one and the one MSX-MIDI gave them.

Read about them in the MSX2 Technical Handbook and the MSX-Datapack, volumes 2 and 3.

K.7 PORTS.INC

The I/O ports, from `IO_MUSREG` (07Ch) to `IO_MAPSEG` (0FCh), and a few memory addresses that behave like ports, such as those of the MSX-MUSIC and MSX-AUDIO cartridges. The MSX has no standard names for its ports, so these names are Tatara's own: a port is `IO_` and a name, and is written with three hex digits.

Read about the ports in the MSX2 Technical Handbook, the MSX turbo R Technical Handbook and the MSX-Datapack, volumes 2 and 3.

K.8 ASCII.INC

The control codes, from `CHR_NUL` (0) to `CHR_DEL` (127), with `CHR_SPACE`, `CHR_EOF`, the end of a text file, and the two quotation marks. The values are decimal, unlike those of the other files, and each comment says what the MSX does when the code is printed.

Read about them in the MSX-Datapack, volume 2.

K.9 EXTBIO.INC

The extended BIOS: the device numbers, the function numbers and the offsets in the jump tables of the devices that answer through it, such as MSX-DOS2's mapper routines, the RS-232C, MSX-AUDIO, MSX-MIDI and MSX-JE. Every name starts with `EXTBIO_` and the name of the device. The file ends with the FM BIOS of MSX-MUSIC, whose names start with `FMBIOS_`: these are addresses in the MSX-MUSIC ROM, which is in another slot.

Read about them in the MSX-Datapack, volumes 2 and 3.

K.10 Further reading

These are the books and documents the descriptions above refer to. The MSX-Datapack and the MSX turbo R Technical Handbook were published in Japanese only.

MSX2 Technical Handbook (ASCII). The BIOS, the SUB ROM, the work area, the hooks, the VDP and the I/O ports of the MSX2. It was also published in English.

MSX-Datapack (ASCII), three volumes. The complete reference, with the BIOS and the SUB ROM in volume 2, and MSX-DOS2 in volume 3. The include files were made from it.

MSX turbo R Technical Handbook (ASCII). What the MSX turbo R added: the R800, the CPU switching, the PCM and the MIDI.

MSX-DOS2 Function Specification and **MSX-DOS2 Program Interface Specification** (ASCII). The functions, their registers and the error codes, and how a program runs under MSX-DOS2.

APPENDIX L

Glossary

This glossary lists the words the manual defines, and some names that it uses without defining them. Each entry ends with the section where the word is introduced, and names related entries after *See also*.

absolute Said of a value that is complete when TATARA assembles it, such as a number or a name defined with `EQU`. Section 8.6. See also *relocatable*, *external*.

absolute path A filename that starts with a drive or with `\`, and so says exactly where the file is. Section 13.4. See also *relative path*.

absolute segment The segment, selected with `ASEG`, in which every address is the one written and TANREN moves nothing. Section 10.2. See also *segment*.

addend The part of a value that is already known when TATARA writes the object file. It is stored in the hole, and a fixup says what to add to it. Section F.1. See also *fixup*, *hole*.

address The number of a memory location. A label stands for the address of the first byte its line produces. Section 2.2.

archive A file that holds several files packed together. Tatara is distributed as one, `TATARxyz.ZIP`. Section 3.1.

argument A value written after the name of a macro when it is used. It takes the place of the matching parameter in the body. Section 16.1. See also *parameter*.

assembling Turning each line of a source file into the bytes of its machine code. Section 2.2.

banner The lines a tool prints first: its name and version, the copyright and the web address. Section 5.4. See also *summary line*.

BDOS The entry at 0005h through which a program calls the functions of MSX-DOS. `MSXDOS.INC` names it. Section 14.3.

BIOS The routines in the MAIN ROM that a program calls to use the hardware. `BIOS.INC` names their entries. Section 14.4. See also *inter-slot call*.

BLOAD The MSX-BASIC command that loads a machine-code file. Such a file has the extension `.BIN` and starts with a BLOAD header. Section 26.4. See also *BLOAD header*.

BLOAD header The seven bytes at the start of a file for `BLOAD`, which say where it is loaded and where it starts. TANREN writes one with `/B`. Section 26.4.

body The lines between `MACRO` and `ENDM`, which TATARA puts in the place of each use of the macro. Section 16.1. See also *macro*.

build file A batch file, `BUILD.BAT`, that runs the commands that build an example. Section 4.6.

character constant One or two characters between quotation marks, used as a number: their codes. Section 8.3.

code segment The segment, selected with `CSEG`, that holds a program's instructions. Section 2.6. See also *data segment*, *segment*.

.COM file A program file that MSX-DOS loads at 0100h and runs when its name is typed. It is what TANREN writes by default. Section 2.1. See also *program file*.

comment Text for the reader, from a semicolon to the end of the line. TATARA ignores it. Section 6.4.

conditional assembly Assembling some lines of a source and skipping others, depending on a condition TATARA works out while it reads. Chapter 15.

current directory The directory you were in when you typed the command. Section 13.2.

data segment The segment, selected with `DSEG`, that holds a program's variables. Section 2.6. See also *code segment*, *transient data segment*.

defined Said of a name that has a value. For TANREN, a public name is defined by the module that declares it. Section 25.1. See also *referenced*.

displacement The signed number from -128 to 127 in `(IX+d)` or `(IY+d)`, added to the index register. Section 9.2.

entry address The address where a program starts to run, named on an `END` line. TANREN shows it as `entry`. Section 26.5. See also *start address*.

environment variable A name that holds a piece of text, which MSX-DOS2 keeps in memory and programs can read. Appendix I describes the ones the tools use. Section 3.4.

error message The message a tool prints when it finds a mistake, before it stops. Chapter 20. See also *trail*, *internal error*.

expansion The lines that one use of a macro or a repeat block produces. Chapter 16. See also *macro*.

expression A number, a name, or a calculation made of them, written where a number is expected. Chapter 8.

extended BIOS The way a program asks which cartridges offer routines for a device, and finds them. Section 14.6.

external Said of a value that depends on a name defined in another module. TANREN finishes it. Section 8.6. See also *absolute*, *relocatable*, *external symbol*.

external symbol A name that a module uses but another module defines, declared with `EXTRN`. Section 2.3. See also *public symbol*.

field One of the four parts of a line: label, operation, operand and comment. Section 6.2.

first pass The first reading of the source, in which TATARA works out the address of every label and the value of every name. Section 15.2. See also *pass*, *second pass*.

fixup The note in an object file that says what to add to a hole: where a segment is placed, or the value of an external name. Section F.1. See also *addend*, *hole*.

free Said of a segment of the memory mapper that MSX-DOS2 has not given to anything else. Section G.2. See also *memory mapper*.

group A set of variables in a transient data segment, started by a `GROUP` line. The groups of a segment share the same memory. Section 10.6. See also *transient data segment*.

hole A place in the machine code whose value TATARA could not finish. The object file records what belongs there, and TANREN fills it. Section 2.3. See also *fixup*, *relocation*.

hook Five bytes in RAM that the system calls at a fixed moment, which a program can take over. Section 14.6.

immediate value An operand without brackets: the number itself goes into the instruction. Section 9.2.

include file A file that another source reads with `INCLUDE`, as if its lines were written there. Section 13.1.

include guard A conditional around the contents of a file, meant to make it safe to include twice. It does not work in TATARA, because a name defined on the first pass is still defined on the second. Section 13.6.

index half IXH, IXL, IYH or IYL: the high or low byte of IX or IY, used as an 8-bit register by one of the undocumented instructions. Section 9.5. See also *undocumented instructions*.

internal error A message that starts with `internal -`: TATARA has found a fault in itself, not in the source. Section 20.5.

inter-slot call A call to a routine in another slot, made through `CALSLT`. Section 14.4. See also *slot*.

label A name written at the start of a line, which stands for the address of the first byte the line produces. Section 2.2. See also *symbol*.

link file A text file that holds what would otherwise be typed on TANREN's command line. Section 4.7.

linking Joining object files into a program file: placing the modules, filling the holes and writing the result. Section 2.4. See also *relocation*.

listing A printout of the source as TATARA assembled it, with the bytes of each line, divided into pages. Section 5.5. See also *page heading*.

location counter The count of the bytes assembled so far in a segment. A label takes its value, and `$` reads it. Section 10.1.

M80 MACRO-80, Microsoft's assembler, whose source format TATARA follows. On MSX it came in ASCII's MSX-DOS2 TOOLS. Appendix J.

machine code The bytes that the Z80 reads and runs. Section 2.2.

macro A name for a group of lines. Each use of the name is replaced by the lines. Chapter 16. See also *body*, *expansion*.

mapper segment One 16 KB part of the memory mapper. The Z80 sees one in each page. Section G.1. See also *memory mapper*, *page*.

memory mapper RAM divided into segments of 16 KB, of which the Z80 sees four at a time. The tools keep their tables there. Section G.1. See also *mapper segment*.

mnemonic The name people give to an opcode, such as `LD` or `DJNZ`. Section 9.4. See also *opcode*.

module The code of one source file, as it is in its object file. Section 2.3.

MSX-DOS1 The first MSX-DOS. The tools do not run under it. Section 1.4. See also *MSX-DOS2*.

MSX-DOS2 The version of MSX-DOS with subdirectories, environment variables and the memory mapper routines, which the tools need. Section 1.4. See also *Nextor*.

name A label, a value defined with `EQU` or `DEFL`, a macro or a segment. Chapter 7. See also *symbol*.

named segment A code or data segment selected by a name after `CSEG` or `DSEG`. Section 10.5. See also *segment*.

Nextor A system that runs the programs written for MSX-DOS2, and so runs the tools. Section H.1. See also *MSX-DOS2*.

object file The file TATARA writes from a source file, with the extension `.TRO`: machine code with holes, and a note for each hole. Section 2.1. See also *module*.

opcode The first byte or two of an instruction, which say which instruction it is. Section 9.4. See also *mnemonic*.

operand What the operation works on, such as `de,msg`: the third field of a line. Section 6.2.

operation The instruction, directive or macro of a line: its second field. Section 6.2.

page One quarter of the Z80's 64 KB: 0000h, 4000h, 8000h or C000h up. Section G.1. See also *mapper segment*.

page heading The lines at the top of each page of a listing: the title, the version, the date, the page number and the subtitle. Section 18.2.

parameter A name in the body of a macro that each use replaces with an argument. Section 16.1. See also *argument*.

pass One reading of the source. TATARA reads it twice. Section 15.2. See also *first pass*, *second pass*.

precedence The order in which the operators of an expression are worked out. Section 8.5.

program file The finished program that TANREN writes, such as a `.COM` file. Section 2.1.

public symbol A name that a module defines and makes available to other modules, declared with `PUBLIC`. Section 2.3. See also *external symbol*.

R800 The processor of the MSX turbo R, which runs the Z80's instructions and a few of its own. Section 9.4.

record One part of an object file: a type byte, a length, and the contents. Section F.1.

referenced Said of an external name that a module uses. Every referenced name must be defined by some module. Section 25.1. See also *defined*.

register code The number of a register in the bytes of an instruction, from 0 for B to 7 for A. Section B.1.

relative jump A `JR` or `DJNZ`, which holds the distance to its target rather than its address. Section 9.3.

relative path A filename that does not start with a drive or with `\`, and is looked for in several places. Section 13.4. See also *absolute path*.

relocatable Said of an address that depends on where its segment is placed, which TANREN finishes. Section 2.3. See also *absolute*, *external*.

relocation Filling the holes of the machine code, once the addresses are known. Section 2.4. See also *hole*.

repeat block A group of lines that TATARA repeats where it stands, with `REPT`, `IRP` or `IRPC`. Chapter 17.

second pass The second reading of the source, in which every value is known and TATARA writes the object file and the listing. Section 15.2. See also *first pass*, *pass*.

segment A part of a program that TANREN places as a whole: the code segment, the data segment, a named segment or the absolute segment. Section 2.6. See also *code segment*, *data segment*.

slot One of the places in the MSX where ROM, RAM or a cartridge can be. Each page of the Z80's 64 KB is set to one slot. Section 14.4. See also *inter-slot call*.

source file A text file in assembly language, normally with the extension `.AS`, that TATARA reads. Section 2.1.

start address The address named by the operand of `END`, where the program begins to run. Section 12.6. See also *entry address*.

statement A line that holds an instruction, a directive or the name of a macro. Section 6.1.

string Text between quotation marks, stored as one byte for each character. Section 11.2.

summary line The last line a tool prints, which says where the source ended or what was written. Section 5.4. See also *banner*.

symbol A name that stands for a number. Section 2.2. See also *label*, *name*.

symbol table The list of every name a source defined, with its value. `/S` prints it. Chapter 19.

trail The lines after an error message that show the include files, macros and repeat blocks that led to the line. Chapter 20. See also *error message*.

transient data segment A named data segment whose groups of variables share the same memory. Section 10.6. See also *group*.

.TRO file An object file, as TATARA writes it and TANREN reads it. Appendix F describes the format. Section 2.1. See also *object file*.

undocumented instructions Z80 instructions that Zilog never published but that every Z80 runs. TATARA accepts SLL, the index halves, IN F,(C) and OUT (C),0. Section 9.1. See also *index half*.

usage screen The summary of the command line that a tool prints with `/?`. Section 5.4.

variable symbol A name defined with `DEFL`, whose value can change. Section 12.4.

word Two bytes, low byte first, as the Z80 stores every 16-bit value. Section 11.3.

work area The part of RAM from 0F323h up where the system keeps its variables. Section 14.6. See also *hook*.

Index

.LALL (directive), 132, 133, 234, 235, 282, 287
.LFCOND (directive), 134, 282, 287, 288
.LIST (directive), 132, 233, 282, 288, 350
.SALL (directive), 132, 133, 235, 282, 288
.SFCOND (directive), 134, 282, 288
.TFCOND (directive), 134, 282, 288, 350
.XALL (directive), 132, 235, 282, 288
.XLIST (directive), 132, 233, 282, 288
/? (option), 36, 38, 161, 162, 367
/B (option), 159–162, 193–195, 197, 198, 201, 253, 258, 320, 362
/C (option), 4, 36, 46, 51, 52, 97, 107, 118, 140, 188, 189, 191, 217, 239, 240, 242, 244, 245, 319, 327, 330, 352
/D (option), 164, 320
/D: (option), 160, 161, 178, 181, 318
/F (option), 37, 137, 142, 144, 153
/H (option), 37, 137, 142, 145
/L (option), 36, 38, 39, 127, 142, 149, 232, 241
/M (option), 37, 137, 142, 144, 145, 153, 161, 176, 183, 185, 188, 190, 191, 215, 222
/O (option), 164, 320
/O: (option), 26, 160, 161, 168, 172, 193–195, 269, 273, 345
/P (option), 36, 38, 39, 127, 131, 144, 149, 164, 241, 320
/P: (option), 160, 161, 178, 194, 199, 252, 260
/Q (option), 27, 33, 36, 37, 139, 148, 159, 161–163, 198, 204, 206, 214, 215, 222, 249, 258, 267, 273, 274, 314, 320
/R (option), 161, 325, 330–332
/S (option), 36, 52, 137, 141, 142, 222, 241, 353, 367
/V (option), 21, 22, 36, 37, 161, 162, 275

absolute, 59, 186, 312
absolute path, 90, 172
absolute segment, 71
addend, 326
address, 9
archive, 17
arguments, 112
ASEG (directive), 70, 129, 194, 248, 252, 257, 260, 280, 282, 350, 361
assembling, 8

banner, 37
BLOAD header, 195
body, 112
build file, 27

character constant, 56
code segment, 13
comment, 46
COND (directive), 104, 282, 289, 350
Conditional assembly, 103
CSEG (directive), 13, 70, 72, 280, 282, 314, 350, 362, 365
current directory, 89

data segment, 13
DB (directive), 55, 57, 77–80, 113, 129, 177, 260, 279, 283, 289, 311, 350, 352
DC (directive), 77, 79, 279, 283, 315, 350

DEFB (directive), 80, 283, 289, 350
defined, 184
DEFL (directive), 50, 51, 57, 83, 84, 122, 140, 141, 240, 280, 283, 287, 288, 315, 317, 350, 353, 365, 367
DEFM (directive), 80, 283, 289, 350
DEFS (directive), 71, 80, 283, 289, 350
DEFW (directive), 80, 283, 289, 350
displacement, 63
DS (directive), 71–73, 77, 80, 177, 179, 180, 199, 200, 224, 260, 279, 283, 289, 350
DSEG (directive), 13, 70, 72, 280, 283, 314, 350, 362, 365
DW (directive), 55, 77–80, 177, 279, 283, 289, 312, 315, 350

ELSE (directive), 103, 108, 109, 134, 281, 284, 285, 288, 289, 315, 350
END (directive), 25, 37, 81, 85, 86, 92, 159, 162, 196, 201, 212, 248, 275, 280, 284, 320, 329, 332, 350, 362, 367
ENDIF (directive), 103, 104, 108, 109, 134, 281, 284, 285, 288, 289, 315, 350, 351, 353
ENDM (directive), 112, 117, 118, 121, 125, 145, 281, 284, 286, 287, 289, 315, 316, 350, 362
ENTRY (directive), 83, 284, 289, 350
entry address, 196
environment variable, 20
EQU (directive), 49, 50, 57, 59, 73, 82–84, 91, 95, 112, 128, 137, 140, 141, 186, 222, 280, 283, 284, 287, 288, 314, 315, 317, 337, 350, 361, 365
error message, 147
EXITM (directive), 111, 116, 118, 121, 125, 281, 284, 289, 315, 350
expansion, 111
expression, 55
EXT (directive), 83, 284, 289, 350
extended BIOS, 101
external, 312
external symbol, 10
EXTRN (directive), 10, 59, 68, 83, 85, 86, 88, 106, 129, 140, 183, 187, 280, 284, 285, 289, 312, 314–317, 350, 351, 353, 363

fields, 44
first pass, 105
fixup, 326
free, 336

GROUP (directive), 73, 76, 280, 283–285, 289, 314, 316, 318, 352, 363
groups, 73

hole, 9
hook, 100

IF (directive), 103, 104, 106, 108, 109, 114, 134, 148, 281, 282, 285, 286, 288, 289, 315, 318, 350
IF1 (directive), 107, 281, 285, 317
IF2 (directive), 107, 281, 285
IFB (directive), 107, 113, 116, 281, 285, 351
IFDEF (directive), 106, 281, 285, 286, 351
IFDIF (directive), 107, 116, 281, 285, 350
IFE (directive), 103, 104, 281, 285, 289, 350
IFF (directive), 104, 285, 289, 350
IFIDN (directive), 107, 116, 281, 285
IFNB (directive), 107, 113, 116, 281, 286, 351
IFNDEF (directive), 91, 106, 281, 286
IFT (directive), 104, 286, 289, 350
immediate value, 62
INCLUDE (directive), 5, 12, 87–90, 92, 93, 95, 150, 169, 233, 234, 263, 265, 280, 286, 288, 315, 316, 344, 345, 350, 353, 364
INCLUDE (environment variable), 352, 354
include file, 87
include guard, 91

index halves, 65
inter-slot call, 98
internal error, 153
IRP (directive), 111, 121–123, 150, 231, 281, 284, 286, 289, 315, 350, 366
IRPC (directive), 111, 121, 124, 150, 231, 236, 281, 284, 286, 289, 315, 350, 366

label, 9
link file, 28, 163, 165
linking, 10
listing, 38
LOCAL (directive), 54, 111, 115, 116, 118, 121, 124, 140, 144, 145, 234, 281, 286, 289, 316, 317, 350
location counter, 57, 69

machine code, 8
MACRO (directive), 112, 117, 118, 144, 148, 281, 286, 288, 314–316, 350, 362
macro, 111
memory mapper, 335
mnemonic, 64
module, 9

named segment, 72
names, 49
Nextor, 339

object file, 7
object files, 325
opcode, 64
operand, 44
operation, 44
ORG (directive), 71, 129, 194, 248, 251, 252, 280, 286, 288, 317, 350
ORG 4000h (directive), 257

PAGE (directive), 129, 131, 135, 282, 287, 289, 314, 350
page heading, 129
pages, 335
parameters, 112
pass, 92, 105
PATH (environment variable), 18, 20, 21, 23, 272, 276, 343, 344, 346
precedence, 58
program file, 7
PUBLIC (directive), 9, 45, 68, 83–85, 88, 139–141, 149, 183, 280, 284, 287, 289, 314–316, 349–351, 353, 366
public symbol, 9

record, 326
referenced, 184
register code, 292
relative jumps, 63
relative path, 90, 172
relocatable, 9, 312
relocation, 10
repeat block, 121
REPT (directive), 111, 121, 122, 145, 150, 231, 281, 284, 287, 289, 317, 318, 350, 366

second pass, 105
segments, 13
source files, 7
start address, 85
statement, 43
string, 78, 329
SUBTTL (directive), 129, 131, 282, 287, 316, 350, 353
summary line, 37
symbol, 8
symbol table, 137

TANREN (environment variable), 5, 21, 169, 171–173, 319, 323, 343, 345, 346
TATARA (environment variable), 5, 18, 20–25, 27, 87, 89, 90, 171, 214–216, 224, 236, 241, 252, 259, 263–268, 273, 274, 286, 317, 322, 343–346, 352, 354
TITLE (directive), 129, 131, 282, 287, 350

trail, 147, 313
transient data segments, 73

undocumented instructions, 61
usage screen, 38

variable symbol, 83

word, 78, 329
work area, 101