Contents

Click on the CONTENT field to go to the relevant section.

CHA	SEC	CONTENT
i		Copyright Notice
ii		Distribution
iii		Disclaimer
iv		Other General Information
v		Installation and Usage
vi		Running Java Tiny Tim
1		Definitions and Terminology
2		Introduction
3		Value Notations
	3.1	Binary (Bin)
	3.2	Hexadecimal (Hex)
	3.3	Converting between Notations
	3.4	Converting Binary Values to Hexadecimal
	3.5	Converting Hexadecimal Values to Binary
	3.6	Using 32-bit Values
	3.7	Using Twos Complement (2s-Comp)
4		The Instruction Set
	4.1	Inside Java Tiny Tim
	4.2	Instruction Syntax
	4.3	The Instructions: A Closer Look - Mnemonic Notation
	4.4	The Instructions: A Closer Look - Machinecode Notation
	4.5	Mnemonic Exceptions and Keywords
	4.6	Helpful Tips: Basic
	4.7	Helpful Tips: More Advanced

5		Memory, Registers and Flags
	5.1	Memory
	5.2	Registers
	5.3	The Instruction Pointer (r6)
	5.4	Flags and the Status Word (r7)
6		The CLI Command Set
	6.1	Command Syntax
	6.2	Command List
	6.3	Dealing with Loading/Saving Error Messages
	6.4	Dealing with Other Error Messages
	6.5	Error Recovery
	6.6	JTT Beans
	6.7	Manual File Manipulation
	6.8	Hints and Tips
7		The GUI Command Set
	7.1	Dealing with Requesters and Error Messages
	7.2	Using Pull-Down Menus
	7.3	Using Pop-Up Menus
	7.4	The Main Window Menu
	7.5	The Help Database
	7.6	Hints and Tips
8		Other Sources of Starting and Further Information
9		The History of Java Tiny Tim
10		Unsolved Errors, Queries and Bug Reports
		Glossary (currently empty)
		Index (currently empty)

JTT Manual: HTML Version 1.1 -- last updated 01/11/01 (MRA)

top | contents | homepage | e-mail your queries

i. copyright

The original Tiny Tim code is (C) Copyright Keele University, Dr. Barry Cook and Dr. Neil White. Special permission was given to Michael Alcock to add its Java/GUI enhancements.

"Java Tiny Tim" and "Java Tiny Tim Mk. II" ("Big Tim") are (C) Copyright Drs. Barry Cook and Neil White, and Michael Alcock, and is based upon the principles of "Tiny Tim", though created by Michael Alcock.

ii. distribution

No part of Tiny Tim or its Java/GUI enhancements, or any relative material thereto, in any form, thereof, may be distributed, changed or split up without the complete written permission of all relative copyright-holders.

The author of Java Tiny Tim (i.e. that commonly distributed with this file) grants that the Java Tiny Tim package, which includes the Java Tiny Tim program files, documentation and all other files used by the Java Tiny Tim program, may freely be distributed, without cost and permission, as long as the following are observed:

1. No charge is made or payment received as a result of the distribution; AND
2. All of the files included in its package are kept together and in their original form and respective directories; AND
3. None of the files included in its package is deleted or changed in any way.

These rules reflect the legal requirements of Keele University and of the wishes of the authors concerned.

iii. disclaimer

The original JTT author, Michael Alcock, does NOT guarantee any part of the Java Tiny Tim Package ("Package"), which includes the Java Tiny Tim binary files and documentation, and any other miscellaneous files used by the Java Tiny Tim programs. Use is at the user's own risk.

No responsibility will be taken for damage or loss of data or equipment as a result, directly or indirectly, of the use or installation of the Package.

The Package is meant as a guide only. No guarantee is given as to the accuracy of this Guide or any other part of the Package, and no responsibility can be taken for damage or loss of data, equipment or hair as a result of the use thereof. Once again, use is at the user's own risk.

iv. other general information

* Also contained in the 'docs/' directory is the
  Version History file, "history.doc", which lists
  the main enhancements to the project, from birth
  to the present day. It is in MS Word format.
* For details of installation, see Section V.

v. installation and usage

To install WinZip-packed Java Tiny Tim onto a Windows ('95+) PC:

1. Place the "jtt.zip" WinZip file in the directory you want Java Tiny Tim.
2. Unzip the "jtt.zip" file using WinZip (download it from www.winzip.com).

To install Java Tiny Tim unpacked on a PC or other system:

1. Create the directory where you want Java Tiny Tim.
2. Move all Java Tiny Tim files and directories into the directory you created.

NOTE: Around 2MB should be spared for Java Tiny Tim and its files.
NOTE: To use Java Tiny Tim, you will need the 'java' program in the 'C:\Windows\' directory, and the Java package installed, or Java Tiny Tim will not run.

COPYRIGHT NOTES:

"WinZip" is (C) Copyright 1991-2000 WinZip Computing, Inc.

"Windows" and "MS-DOS" are (C) Copyright Microsoft Corp.

"SunOS" and "Java" and Java technology are (C) Copyright Sun Microsystems, Inc.

vi. running Java Tiny Tim

To run Java Tiny Tim:

```
1. Open a CLI prompt (MS-DOS in Windows; a Terminal Console
   in Unix/SunOS; a CLI or Shell in AmigaOS)
2. Go into Java Tiny Tim's home directory (e.g. c:\java\jtt)
3. Do one of the following:
   a. Type 'java jtt' (without quotes); or
   b. Type 'jtt' (without quotes) to run the batch file.
```

For further information about using Java Tiny Tim's command-line interface, see Chapter 6 of this Manual.

COPYRIGHT NOTES:

"Windows" and "MS-DOS" are (C) Copyright Microsoft Corp.

"Java" and Java technology is (C) Copyright Sun Microsystems, Inc.

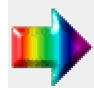
1. Definitions and Terminology

Throughout this Manual, and in JTT itself, various words will be used. A more extensive list is contained in the glossary at the end of the Manual. However, it is important to know the meaning of some terms before we start. Some of those not covered under this section will be explained as we go through. Here's what the main ones mean (in alphabetical order):

Please Note
All definitions given in this Manual and in the JTT package are relative to their use therein, and do not necessarily reflect generally or specifically accepted terminology.

Word/Phrase	***Explanation***
^	'To the power of'. 2^4 = 2 to the power of 4=decimal 16.
Assembler	The program that compiles (assembles) mnemonic assembly code into machinecode.
Assembly	The low-level programming language that Java Tiny Tim emulates. In it, one instruction corresponds to exactly one operation that the processor is capable of carrying out.
Binary	Binary is a way of representing values as 0s and 1s, and is used by computers to store information. The values 0 and 1 are as in decimal. However, 2 is '10', 3 is '11' and so on. Read Section 3: Value Notations for more.
Bit	A binary digit; a 0 or 1 that makes up a binary value.
C/C++	The programming languages from which Java emerged.
CLI	The command-line interface, which is the text screen that the user uses to control JTT.
Code	The sequence of text that represents the instructions that the computer is to carry out.
Command	A string of text that tells JTT's CLI to perform a particular task, such as showing help or loading a file.
Compile; Compiler	To compile means to 'change' code from mnemonic form to its machinecode equivalent. The program that does this is called the compiler.
Constant	A constant is an absolute value. In JTT, values input directly or held in data memory are constants, whilst sums (e.g. 3+4) are not, as the final value has not yet been calculated.
Data	A value or values to be read, stored or manipulated.
DWord	A DWord, or double word, is a 32-bit value.

Exception	As in the phrase 'exception to the rule', an exception is a circumstance in which the rule that normally governs an operation is not obeyed, thereby giving way to a special set of rules. Exceptions can be errors, but they can also simply be special circumstances as a result of a calculation.
GUI	The graphical user interface, which is the series of point and click windows and images that the user uses to control JTT.
Hardware	Since JTT is wholly software, this term applies to those parts of the JTT program that mimic (i.e. would otherwise be implemented in) hardware. Hardware itself is anything computer-related that is not software, i.e. physical devices such as the keyboard, monitor and mouse.
Home Directory	The place where the JTT program file, preferences file and 'docs/' directory is installed.
Hex; Hexadecimal	Hex is the system of denoting values in base-16. The digits 1-9 are represented as in decimal, but 10 becomes 'A', 11 becomes 'B', and so on up to 15 ('F') .It is a shorthand method of representing binary values. Read Section 3: Value Notations for more.
Instruction	One of the set of eight operations that the JTT emulator is capable of doing, such as Subtract and BitSelect.
Java	The third-generation object-orientated programming language that was used to program Java Tiny Tim.
JTT	Java Tiny Tim, the program we're all talking about.
Program	The complete sequence of instructions that together perform a particular task.
Software	Any application, program, piece of code or instruction.
TT	Tiny Tim, the original from which Java Tiny Tim came.
User	The person using/programming in JTT.
Value	A value is a number, which in JTT can normally be represented in decimal, hexadecimal or binary format.
Word	Another name for a binary value that is 16 bits wide (long).



top contents homepage e-mail your queries

2. Introduction

Please Note

"Tiny Tim" was the title of the original project, from which Java Tiny Tim emerged. 'Java Tiny Tim' was rewritten in Java (Tiny Tim was originally written in C), allowing enhancements that Tiny Tim would not. It is no way intended to be a first-step towards using Java specifically, only towards using modular languages and assembly.

You may occasionally see the term "Big Tim", which is the nickname for Java Tiny Tim. For simplicity, the original title (Java Tiny Tim, or "JTT") will be used in this Manual. For more information on Tiny Tim's background, see Chapter 8.

Java Tiny Tim (JTT) may be called a 'virtual machine'. Although it consists entirely of software, it aims at emulating the basic functions of a computer's processor.

Its purpose is to develop an assembler package aimed at people wanting to learn low-level programming language concepts, i.e. where each instruction corresponds to only one processor instruction. It focuses on the principles of:

- Computer hardware and architecture concepts
- Low-level programming languages
- Operating systems
- Mathematical concepts — for example, boolean algebra

It includes eight assembly instructions, a compiler and basic facilities s such as a command-line interface (CLI), 8 registers, interrupts and error management.

JTT is aimed primarily at a general adult audience, especially computer science undergraduates at university, but also allows those with little or no experience in using assembler to easily develop skills in it. Some amount of experience in other programming languages, however, such as a BASIC, should be considered a significant benefit.

It also allows the user to become familiar with modular programming languages, such as Java, by allowing programs to jump to other stored files as well as other parts of the instruction memory. Its special features include:

- A Graphical User Interface (GUI) front-end
- Loading and saving
- Memory management
- Pseudo-code and machinecode representations of code
- Extensive error-trapping and data recovery
- "JTT Beans" (JTTBs), which can be 'mail-merged' into JTT code
- A cut, copy and paste facility;
- A macro enabling the user to jump to JTT code within another file;

- An on-line help system.

The error-trapping and recovery system involves a 'dumping' of JTT instruction and data memory and registers to a file before exiting (if possible), so that upon loading again the previous session is resumed automatically and without user prompting.

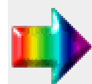
The on-line help system is accessible via the CLI (using the 'h' command), given in straightforward text format, and the GUI (using the GUI's menu), in the form of a database. In the GUI's help system, the user can type in key words, and an index is created with a choice of appropriate help pages.

The existing JTTBs provide a series of methods, written in JTT, are for use in:

- Providing examples with which the user can become familiar with JTT's operations
- Providing hands-on examples as extensions to the help provided on-line (i.e. within the GUI) and in the Manual
- Extending programs, or parts of programs without having to type out repeatedly more common functions (modularization)
- Promoting the modular aspect to most programming languages

Please Note

To avoid ambiguity, in the world of JTT the term 'instruction' is given to the eight operations of JTT's emulator (the instruction set), whilst the term 'command' is given to any of the operations used within JTT's CLI. The reader is reminded of the distinction throughout the Manual. For more information on the two terms, see Section 1.



3. Value Notations

Although JTT usually does not mind which you use, you must say which you are using by affixing its symbol, if any, with the data. The table below shows the notations and identifiers. These symbols are used through the Manual to avoid ambiguity.

JTT supports three value notations (representations): decimal (dec), binary (bin) and hexadecimal (hex). The latter two are elaborated upon in this section, and decimal is just mentioned (since it is expected that you know how to use this).

Please Note
When entering values in any notation, it is not necessary to have a certain number of digits. E.g., (decimal) 00012 can be written 12. However, there are limits to the value that can be entered. JTT is predominantly 16-bit, so for each value entered, the minimum is 0 and the maximum 2^16 (i.e. a range of 0 to 65535 (decimal), $0 to $1111 1111 1111 1111 (binary) or 0x0 to 0xffff (hex).
Sometimes, for clarity, large decimal values are separated by series of commas, and binary and hex values by spaces. However, it should be noted that they should be typed without such spacers, as if a fluent value, or the result will be a syntax error. Of course, decimal 00012 equals 12, whilst 12000 does not, since trailing digits denote ever increasing values. In the same way, remember that binary $001 equals $1 but not $100, and hex 0x0f equals 0xf but not 0xf0, for example.

Notation	Template	Example	Details
Decimal		123	This is the default, no symbol needed.
Binary	$	$01011	The dollar sign precedes binary values.
Hex	0x	0xff	'0x' precedes hex values. It is an old convention, used in many languages.

3.1 Binary (Bin)

Binary values are base-2. So, whilst decimal is base-10 (after 9, digit is added, lowest digit reset to 0), in binary, after ‘1’ a digit is added and the lowest reset to 0. The table below gives examples. Note that each time a binary digit (called a 'bit') is added, the corresponding value doubles.

Dec.	Bin.
0	$0
1	$1
2	$10
3	$11
6	$110

Dec.	Bin.
15	$1111
16	$1 0000
31	$1 1111
32	$10 0000
63	$11 1111

Dec.	Bin.
255	$1111 1111
512	$10 0000 0000
1024	$100 0000 0000
2048	$1000 0000 0000
4096	$1 0000 0000 0000

7	$111	64	$100 0000	8192	$10 0000 0000 0000
8	$1000	127	$111 1111	65535	$1111 1111 1111 1111

Hex values are base-16. Per digit, 0 to 9 are as with decimal. However, 10 becomes A, 11 becomes B and so on up to 15, which becomes F. After that, a new digit is added. Hex is a shorthand for binary. Both are easily compatible, and hex is four times more compact (due to the fact that each digit holds 2^4 times the amount). Hex is widely used in HTML and sometimes in heavily mathematical programs in other languages although supported by most programming languages. The table below gives some examples of its use and the conversion between decimal and hex.

3.2 Hexadecimal (Hex)

Please Note
Hex letters are not case sensitive. However, lower-case is conventional.

Dec.	Hex.
0	0x0
1	0x1
9	0x9
10	0xa
11	0xb
12	0xc
13	0xd
14	0xe
15	0xf

Dec.	Hex.
16	0x10
17	0x11
18	0x12
255	0xff
256	0x1 00
4095	0xfff
4096	0x10 00
65534	0xfffe
65535	0xffff

Here are some examples of associated decimal, binary and hex values, which are within JTT's range. You may use them for reference and study them to f familiarise yourself with the similarities of each notation. Although not used much here, it is useful generally to know that a 16-bit value is known as a 'Word', and a 32-bit value a 'DWord', or double word.

Note that binary doubles with each affixed digit, and hex multiplies by 16. Hence, for example, 0x10=1^16=16, and 0x100=2^16=256.

3.3 Converting between notations

Dec.	Bin.	Hex.
0	$0	0x0
1	$1	0x1
2	$10	0x2
7	$111	0x7
8	$1000	0x8
15	$1111	0xf
16	$10000	0x10
31	$11111	0x1f
32	$10000	0x20

Dec.	Bin.	Hex.
63	$11111	0x3f
64	$100000	0x40
255	$1111111	0xff
256	$10000000	0x100
4095	$111111111111	0x3ff
8191	$1111111111111	0x7ff
16384	$11111111111111	0xfff
32767	$111111111111111	0x7fff
65535	$1111111111111111	0xffff

3.4 Converting Binary values to Hexadecimal

Converting between binary and hex is easy. To convert binary to hex:

1. Split up the binary value into fours from the right. If there are leftover bits on the left, prefix 0s to make up a group of four.
2. Each group will be between 0 and 15, so can be converted to a single hex letter. Do this for each group.
3. Place the letters created together in order to give the total hex value.

See the table below for an example.

	Binary Value:	$11010111011110
Step 1:	Binary Group:	0011 0101 1101 1110
Step 2:	Equivalent Hex:	3 5 d e
Step 3:	Hex Value:	0x35de

3.5 Converting Hexadecimal values to Binary

To convert hex to binary:

1. Split up the hexadecimal value into separate, single digits.
2. For each digit, work out the corresponding binary value, which will be between $0000 and $1111. If the value has less than four digits, prefix 0s to make up.
3. Place the groups of four bits in order, to give the total binary value.

See the table below for an example.

	Hex Value:	0x35de
Step 1:	Hex group:	3 5 d e
Step 2:	Equivalent Binary:	0011 0101 1101 1110
Step 3:	Binary Value:	$11010111011110

Addresses or registers that are coupled to be used with 32-bit values *must* be juxtaposed from an initial even address. That is, an even address (i.e. an address whose value is even) must be coupled with an odd address, and an even register must be coupled with an odd register. This syntax is important (and indeed necessary) in assembly languages, as it automatically rules out illegal memory values to a large extent.

3.6 Using 32-bit Values

Please Note
32-bit values are also called 'double' values. In JTT, the technique can be applied to data memory and registers.

It was said earlier that JTT is predominantly (but not wholly) 16-bit. This is because JTT can computing 32-bit numbers by pairing consecutive data memory addresses, and treating each pair as one value. Let’s take an example. Say memory address 'A' contains the value (decimal) 15, and address 'B' the value (decimal) 5. Value 'AB' is the result. This is illustrated and elaborated in the table below.

Addr	Dec.	Bin.	Hex.
A	15	$0000000000001111	0xf
B	5	$0000000000000101	0x5
AB	1,966,053	$0000 0000 0000 1111 0000 0000 0000 0101	0xf005

You should note that, in the 16-bit to 32-bit conversion (i.e. both Words and DWords), values are treated as binary and ALL 16 bits are taken into account. These two values are then juxtaposed together to make the 32-bit value. As a result, (decimal) 15+5 is not 20 as you would expect. Instead, it is 1,966,053.

When dealing with 32-bit numbers, the number range obviously change. The range of acceptable values (excluding in twos-complement - see Section 3.7) are shown in the table below. The maximum values are split into groups for readability.

Notation	Min.	Maximum Range
Decimal	0	4,294,967,295
Binary	$0	$1111 1111 1111 1111 1111 1111 1111 1111
Hex.	0x0	0xffff ffff

3.7 Using Twos Complement (2s-comp)

Please Note
Those not at all familiar with the basics of twos complement mathematics are advised to skip the twos-complement part of this section until they are more familiar with JTT.

The following operations can also be applied to 32-bit numbers (i.e. DWords).

You will no doubt have noticed that all values that JTT appears to support are positive. However, this is not strictly the case. To use negative values, you must treat them as 2s-comp values.

With negative numbers, the leftmost bit of the 16 bits is used as the 'sign' bit. This is set to 0 if the value is positive, or 1 if the value is negative. Thus, the maximum range of values in 2s-comp is 2^15, or (decimal 32,767 or hex 0x7fff). The whole range is not lost, however, because when the sign bit is set, there is a maximum negative range of (-2^15)+1, or (decimal) -32,768. One is deducted from the negative total to take into account zero, midway up the range.

To convert a binary value between positive and negative, the following two- point rule is employed:

1. NOT the binary value (i.e. convert each 0 to 1 and vice versa).
2. Add one to the total.

It should be noted that the Zero and Negative flags are changed accordingly, and tell you whether the number is positive, negative or zero. The following table clarifies these points:

Dec.	Binary
0	$0000 0000 0000 0000
1	$0000 0000 0000 0001
2	$0000 0000 0000 0010
3	$0000 0000 0000 0011
4	$0000 0000 0000 0100
...	...
65532	$1111 1111 1111 1100
65533	$1111 1111 1111 1101
65534	$1111 1111 1111 1110
65535	$1111 1111 1111 1111

Dec.	Binary
-0	$N/A
-1	$1111 1111 1111 1111
-2	$1111 1111 1111 1110
-3	$1111 1111 1111 1101
-4	$1111 1111 1111 1100
...	...
-65532	$0000 0000 0000 0100
-65533	$0000 0000 0000 0011
-65534	$0000 0000 0000 0010
-65535	$0000 0000 0000 0001

Each positive value (e.g. (decimal) 40) therefore has its relative negative counterpart (e.g. (decimal) -40), which can be reached by the same two-point rule.

Consider the rule. It can be seen in the table above that every negative number is the logical opposite (NOT) of its previous positive counterpart. For example, the binary representation of -4 is the NOT of +3, and -65534 is the NOT of +65533. Notice also that there are negative zero is not catered for, because it is addressed as positive zero. Therefore, -65535 minus 1 = +0.

You can decide whether to treat a 32-bit value as 2s-comp or not within your JTT program by using or ignoring these protocols (and using your brain) accordingly. JTT itself is not capable of distinguishing between positive and negative values, and care should be taken to treat a value in the correct way (usually be employing use of the negative flag).

Please Note
Although JTT does not directly aware of the use of 2s-comp, flags 4-7 of the Status Word do cater for it indirectly, by assuming that the last result was to be used in a twos-complement arrangement by the user.
Out of interest, another way of performing 2s-complement is to decrement the value by 1 before performing logical-OR on each bit.

4. The Instruction Set

JTT has eight instructions, which together makes up JTT's mnemonic language (see later). Each has its own specific low-level function, and can be used in conjunction with each other. As with a Turing machine, almost any task can be accomplished with the correct sequence, and number, of any of these eight instructions. The word 'almost' is used here because JTT only has a limited amount of memory space. Set to 4,096 at default, this means that JTT can hold a maximum of 4,096 instructions (plus 4,096 data elements) at any one time.

Please Note
The instructions work in conjunction with memory, registers and flags. To learn about these in more detail, see Chapter 5 - Memory, Registers and Flags.
Do not confuse JTT's instruction set with the CLI's commands. The former is the series of eight instructions the emulator understands to perform operations within a JTT program. The latter are the operations of the CLI and GUI that the user can use to do things not related directly to JTT programs, such as quitting and displaying help. The functions of the CLI and GUI are dealt with in Chapter 6 and Chapter 7 respectively.

Each instructions, numbered (decimal) 0 to 7, are explained below, including the machinecode and mnemonic format of each and the limitations of use. The numerical result from all instructions (whether calculating new data or moving existing data) is placed in data memory at the given address.

No.	Name	Function
0	Load Constant	Puts a datum constant directly into data memory
1	BitSelect	Selects a given bit of a particular register
2	Add	Adds two values together
3	Subtract	Subtracts the second value from the first
4	Logical NAND	NANDs two values together
5	Rotate Left	Shifts the binary value once to the left
6	Fetch from Memory	Gets a value from data memory
7	Store to Memory	Puts a value into data memory

JTT's architecture supports 16-bit Words, and each has its own function. Some work on their own independent of other operations. Others work in groups that change or have different functions depending on the instruction under which they are influenced. They are shown below in the table.

4.1 Inside JTT

Bit No.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Name	T	I	I	I	C	D	D	D	A	A	A	A	B	B	B	B

- The TRAP instruction bit (T, to the very right) allows additional functions to be added to JTT. However, this can largely be ignored for now.
- The three INSTRUCTION bits (I, to the right of T) hold the unique identifier of the instruction to be carried out. Each of the eight instructions has its own I-bit code: Instruction 0 is $000, up to instruction (decimal) 7, which is $111.
- The CONDITIONAL bit (C) determines whether the instruction is to be carried out only if the Q(uery) flag is set (yes if set to 1, no if otherwise).
- The four DEST bits (D, to the right of the C bit) hold the destination where the result of the instruction will be placed. If the last D-bit (bit-11) is set to 0, then the destination is a register. The register number, 0 to (decimal) 7, is held in the leftover three D-bits, as with the three I-bits above.
- The bits left over, the rightmost eight, are used differently by each instruction. Instruction 0, 'Load Constant', takes A and B as one.
- Instructions 1 to 7 split these last 8 bits into two groups: A (the source) and B (the destination).

Please Note
The TRAP bit (T) is never set to 1 during the use of one of the standard eight instructions. However, two additional implemented instruction make use of the TRAP expansion facility, *HALT*, occurs when all 16 bits are set to 1. In other words, the machinecode instruction $1111 1111 1111 1111' is *HALT*, which stops the program. *NULL* ($1111 1111 1111 1110) signifies a memory address that has not yet appropriated an instruction. More on these later.

Each of the bits can be set to 1 (active) or 0 (inactive). Thus, every possible instruction combination can be represented by a code of 16 bits (i.e. a Word). This is directly recognisable by the processor, or in the case of JTT, by the emulator.

Such a binary code, for these reasons, is called a machinecode language. The representation that is immediately recognisable to humans by its use of English words, is termed mnemonic code. The user can program using either convention. An explanation of each instruction, in both formats, is given in the sections below.

4.2 Instruction Syntax

The instructions will be referenced, in this Manual and in JTT itself, using a specific format. The format is simple, and avoids some ambiguities that may crop up. It is useful to learn the syntax before continuing, and the instructions used as examples will be used in Section 4.2.

1. Necessary parts of an instruction are placed in pointed brackets (< and >). So, for example, the instruction 'r0=0x20' is referenced as <rx>=<value>.
2. Required variables are denoted as x, y and z, which stand for register numbers or register bit numbers (see above example), or <value> which stands for any value given in decimal, binary or hex format (with appropriate identifier prefix).

4.3 The Instructions: A Closer Look - Mnemonic Notation

Please Note
Some of the instructions below may appear to be unnecessarily restricted in their use. Section 4.4 provides the reasons why. However it may be wise to leave this section until you are experienced with using Mnemonic instructions below first.

Template: Instruction 0 - Load Constant

```
<rx>=<value|ry>;
```

There are two uses of Load Constant. The first loads constant value <value> into register <rx>. The second loads the value contained in register <ry> into register <rx>. The constant, <value|ry> may thus be:

- Any 7-bit value (i.e. decimal 0 to 127, $0 to $111 1111, 0x0 to 0x7f)
- The contents of a register

The reason why only 7-bit constants are accepted is explained in Section 4.4. Only a constant value has this restriction; the contents of a register can be 16 bits wide, as normal. To load constants from and to data memory addresses, use fetch from memory and store to memory respectively. Examples of Load Constant:

- To load constant value 0xf into r4: *r4=0xf;*
- To load value of r6 into r0: *r6=r0;*

Template: Instruction 1 - BitSelect

```
<rx>=<ry|value> [<rz|value>];
```

With BitSelect, it should be remembered that the square brackets ([and]) are typed in as part of the instruction, and do not signify optional instruction arguments.

This instruction places a given bit of a given value (or value inside a register) and places it in bit 0 (the least significant bit, to the right) of the destination register. Its major purpose is to test the states of the flags in r7. Since bit 0 of r7 is the Query flag, which can directly be influenced by the JTT programmer, the other bits, such as Negative and Zero, can be placed into Query in order to use the Comparison Instruction Extension (see Chapter 5.4).

It can also allow the programmer to perform tests on the values of registers, by examining the settings of their bits without having to use the flags in r7. This way, you can use memory spaces and registers as your own custom flags amongst others.

The destination must be a register. This can, and usually does include r7. The next value, <ry|value>, can either be one of the registers from which to take the value to BitSelect, or a constant to BitSelect.

The last value is placed inside square brackets, and is the one that designates which bit to take out.

Consequently, it can be a constant between 0 and 15, a register, or one of the flag names. The constant corresponds to bit 0 (the least significant, on the right), bit 15 (the most significant, on the left) or any bit in- between. If a register is given, the four least significant bits are red and the corresponding value is used.

If any of the eight register names is given, it must be a lower-case, single character from those outlined in Chapter 5.4. Examples of its use follow.

```
Example 1: Place the Zero flag into the Query flag for testing.
Solution:  r7=r7[z];

Example 2: Place bit 4 of r5 into r0.
Solution:  r0=r5[4];

Example 3: Place the bit of r1, as designated by r2, into r3.
Solution:  r3=r1[r2];

Example 4: Place the Carry bit into r0.
Solution:  r0=r0[c];
```

Template: Instruction 2 - Add

```
<rx>=<value|ry>+<value|rz>;
```

This simply adds two values, <value|ry> and <value|rz> together, and places the result into register <rx>. The rules for its use are that each constant, <value|reg>, must be:

- Any 7-bit value (i.e. decimal 0 to 128, $0 to $0111 1111, 0x0 to 0x7f)
- The contents of a register (which may, of course, be 16 bits long).

Thus, add can be used as Load Constant except that:

- This is inefficient, since two values are always added together
- The maximum constant that can be used is 254 (127+127)

Memory addresses cannot directly be accessed, so 'Fetch from Memory' and 'Store to Memory' should be utilised.

If the result is larger than a 16-bit value, as may happen if two registers are added together, it will be truncated to the least significant 16 bits and the carry bit will be shown. It is therefore your job to check for such values (via the carry flag) and deal with them as necessary. This also means that the larger the value, the more information is lost and it becomes less accurate. Two examples are:

- To add $1001 0111 to the value in r0 into r1: *r1=r0+$10010111;*
- To add together r3 and r4 and place the result into r6: *r6=r4+r3;*
- To multiply r0 by two and place the result back into r0: *r0=r0+r0;*

Template: Instruction 3 - Subtract

```
<rx>=<value|ry>-<value|rz>;
```

This simply subtracts the second value (<value|rz>) with from the first (<value|ry>), and places the result into register <rx>. The rules for its use are identical to add (above). The comparative examples to add are:

- To take $1001 0111 from the value in r0 into r1: *r1=r0-$10010111;*
- To add minus r3 from r4 and place the result into *r6: r6=r4+r3;*
- To use r0 to get the value zero and place the result back into r0: *r0=r0-r0;*

Please Note
The contents of the memory address which is the addition of the two given values is loaded. It is not the case that the memory address of the first is added to the constant of the second. For example, in the instruction r0=memory[r5+20] , where r5 contains 1, r0 does not become (the contents of address 1)+20. Instead, it becomes (the contents of memory address 21).

Template: Instruction 4 - Logical NAND

```
<memory[<value|rx>+<value|ry>]>=<value|rz>;
```

As defined by De Morgan's Laws, any operation using AND, OR, NOT or XOR can be achieved using the appropriate number and combination of NAND. This stems from electronics in the use of logic gates, and thus may be somewhat a misnomer for some readers. However, the basics are simple, and relate to boolean logic.

Consider that two values, A and B, can be true (1) or false (0). The result is also either true or false, but it is determined by the state of A and B. AND is used when the result is true only when both A AND B are true. With OR, the result is true if either A OR B are true, or if both are true. NOT causes the result to be true if both A and B are false (NOT true), and is therefore the opposite of AND. XOR, or 'exclusive OR', makes the result true if and only if either A OR B are true (but not both nor neither). The following truth tables show each of the four possibilities and outcomes of each:

GATE	A	B	RESULT
AND	0	0	0
AND	1	0	0
AND	0	1	0
AND	1	1	1

GATE	A	B	RESULT
OR	0	0	0
OR	1	0	1
OR	0	1	1
OR	1	1	1

NOT	0	0	1
NOT	1	0	0
NOT	0	1	0
NOT	1	1	0

XOR	0	0	0
XOR	1	0	1
XOR	0	1	1
XOR	1	1	0

Consequently, NAND, or 'not AND', is true when either A or B are true, or if neither are true (but not both). NAND is therefore the opposite of AND. This means that if A and B are NANDed (to make A NAND B), the result is thus with each combination:

0	NAND	0	=	1
0	NAND	1	=	1
1	NAND	0	=	1
1	NAND	1	=	0

When a result is NANDed by itself, as you can see in the table above, true becomes false and false becomes true. Therefore, if 'A NAND B = C', 'C NAND C' produces the same result as 'C NOT C'. Therefore, 'NAND (A NAND B)' = 'NOT (A AND B'). You will have to work out the repetition and order required to make NAND work for the other logic gates, but after NOT, more and more NAND gates will be required.

Template: Instruction 5 - Rotate Left (rol)

```
<rx>=<ry|value>rol<rz|value>;
```

This instruction takes a value (as a constant or the value of a register) and converts it into binary notation. Then, each of the bits are moved ('rotated') to the left and the previous most significant bit (bit 15) is placed into the now vacant least significant bit (bit 0) on the right.

Each time the bits are shifted one place, the result is effectively multiplied by 2, and the number of shifts to carry out is defined in the value (or register value) after the rol instruction.

This provides a computationally uncostly method of squaring values and providing a 'multiply by two to the power of' function. The command can also be used (with the appropriate number of Rotate Left instructions) to the effect of 'rotating' the value right (and thereby square-rooting the result).

As with the add instruction, although the destination has to be a register, the two values to use in the operation (i.e. the value and the number of times to rotate left) can be expressed either as a 7-bit constant or as a register.

If the value becomes too large (i.e. more than 16 bits can hold, which is decimal 65,536 or $1111 1111 1111 1111 or 0xffff), the most significant figure (the leftmost bit) is rotated over to become the least significant figure instead of the 0, and the carry flag is set.

Some examples of its use follow. The first is a straightforward multiplication of a given constant. The second increases a value by a larger power. The third shows what happens when the register runs out of space. The forth increases the value in a register by the power given in another register, and shows that, if the programmer is not careful (see Note for this instruction, below), the actual result may not bear any resemblance to the expected (or required) result.

```
Example 1:  Double the number (decimal) 27.
Manual Working:
1. Decimal 27 in binary notation:          $0000 0000 0001 1011
2. rotate Left once (27 rol 1):            $0000 0000 0011 0110
3. The result is the following in decimal: 54

Example 2:  Calculate 0xfff multiplied by two to the power of 4.
Manual Working:
1. Hex 0xfff in binary notation:           $0000 1111 1111 1111
2. rol four times (0xfff rol 4):           $1111 1111 1111 0000
3. The result is the following in hex:     0xfff0 (decimal 65521)
```

```
Example 3:  Double the result calculated in Example 2.
Manual Working:
1. Decimal 65521 shown in binary:          $1111 1111 1111 0000
2. Rotate left once:                       $1111 1111 1110 0000
3. Place the prev. rightmost bit on left:  $1111 1111 1110 0001
4. The result is the following in hex:     0xffe1 (decimal 65506)

Example 4:  Multiple the value in r0 by the value in r1.
Solution:   r0=r0 rol r1;
```

Please Note

To avoid garbled results from too many rotations, create a loop that tests the carry flag on each rotation. If the carry flag is set, you may either use it as a 17th bit or exit the loop. In either of these cases, if you just want it to left-shift, to double the result, use BitSelect (below) to clear the least significant bit each time.

This command is not fussed whether or not you put spaces between the rol keyword and the two values to the right of the equals symbol. This goes with nand also.

With a loop and a bit of brainpower, this instruction can also be used to imitate a 'rotate right' (ror) instruction. Rotate right is also common in some assembly programming languages.

Template: Instruction 6 - Fetch from Memory

```
<rx>=<memory[<value|ry>+<value|rz>]>;
```

Fetch from Memory stores the contents of a data memory address into register rx. Data memory addresses must always be given as two values to add together, and each value must either be a 7-bit number or 16-bit register, as with the add instruction.

Also as with add, the maximum memory address that can be implied as a constant is 254. However, memory addresses are temporarily stored as 32-bit numbers and are not truncated, allowing JTT to use a maximum of 4,294,967,295 (just over four billion) data and instruction memory a addresses. Some examples follow (next page).

- To load memory address (decimal) 10 into r6: *r6=memory[10+0];*
- To load memory address (r5+24) into r4: *r4=memory[r5+24];*

Template: Instruction 7 - Store to Memory

```
<memory[<value|rx>+<value|ry>]>=<rz>;
```

Store to Memory stores the contents of a register, or a constant value, into the given memory address. See **Fetch from Memory** for usage guide.

- To load r6 into memory address (decimal) 10: *memory[10+0]=r6;*
- To load constant 7 memory address (r5+24): *memory[r5+24]=7;*

Please Note

This instruction can only store the contents of a register (i.e. not a constant value), due to how the machinecode is stored. See Section 4.4 for an explanation why.

4.4 The Instructions: A Closer Look - Machinecode Notation

Please Note

Remember, unlike decimal, binary (and hex) numbers have their most significant figures to the left. As a result, binary numbers appear 'backwards' - that is, the higher bits are given on the left, and bit 0 on the extreme right. Don’t forget to adjust accordingly when dealing with binary values and using the following binary tables.

Only more advanced users of JTT should attempt to insert instructions directly in machinecode notation. Other users should first study and use the mnemonic form of Section 4.1, then move onto this section. It is, however, extremely important to get around to machinecode eventually, since this it is the main goal of JTT to teach users the fundamentals of simple processor operations, and machinecode is at its heart. It is good practice, for learning computer fundamentals and for keeping the brain active.

Instruction 0: Load Constant

'Load Constant' loads a constant value, or the contents of a register, into a register. Due to "hardware" restrictions, this value cannot be smaller than 0 or larger than 2^7, or decimal 127, binary $111 1111 or hex 0x7f. It is evident why if we look at the table below that shows the machinecode representation of this instruction:

Bit No.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	0	0	0	0	C	D	D	D	R	K	K	K	K	K	K	K

The trap bit and instruction bits are all set to 0, and bits (decimal) 0 to 7 (R and K) are treated as one group. Bit 7 denotes whether bits 5 to 7 are registers (if set to 0) or whether bits 0 to 6 should be treated as a direct constant value (if set to 1). Thus, the maximum value can be 7 bits wide, or 2^7, or 16 bits wide if a register is given.

To use this instruction in machinecode is easy. Whilst the value or register to use as the constant is given in bits 0 to 7, are above, bits 8 to 10 contain the three bits designating the destination (D) register number ($000 to $111). Bit 11 is the conditional (C) bit. If the Comparison Instruction Extension (see Chapter 5.4) is used to denote a conditional execution of this instruction, bit 11 is set to 1.

Examples follow.

```
Example 1:  Load constant (decimal) 44 into register 4 (r4).
Solution:
1. The Trap bit and Instruction bits (i.e. bits 12 to 15) are set to 0.
2. The instruction is not conditional, so bit 11 is set to 0.
3. The destination register is r4, so the D bits become $100.
4. The source is not a register, so bit 7 is set to 1.
5. Bits 0 to 6 therefore contain the number (decimal) 44, or $010 1100.
6. The machinecode instruction is thus: $0000 0100 1010 1100, or
   decimal 1,196 or 0x04ac.

Example 2: Load r1 into r0, but only if the query flag is set.
Solution:
1. The trap bit and instruction bits are set to 0.
2. The instruction is conditional, so bit 11 is set to 1.
3. The destination register is r0, so the D bits become $000.
4. The source is a register, so bit 7 is set to 0.
5. Bits 4 to 6 are $001 (r1), and bits 0 to 3 are set to 0.
6. The machinecode instruction is thus: $0000 1000 0001 0000, or
7. decimal 2,064 or 0x0810.
```

Instruction 1: BitSelect

Here, the first value (RAAA) is the value from which to select a bit and the second (RBBB) contains the bit to select. If RBBB is a constant, it can access the least significant half. It a register, the four least significant bits are used to determine which bit to select, from 0 to 15.

Please Note
The bit is put into the least significant bit (bit 0, to the right) of the destination register. The other bits of the destination register remain as they were before the operation.

Bit No.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Name	0	0	0	1	C	D	D	D	R	A	A	A	R	B	B	B

```
Example 1: Place bit (decimal) 6 of r2 into bit 0 of r2.
Solution:
1. The trap bit is set to 0. The instruction bits are set to $001.
2. The instruction is not conditional, so carry is set to 0.
3. The destination is r2, so DDD becomes $010.
4. The source value is in r2, so bit 7 is set to 0 and AAA become $010.
5. The bit to select is (decimal) 6, so bit 7 is 1 and BBB become $110.
6. The machinecode instruction is thus: $0001 0010 0010 1110, or
   decimal 4,654 or 0x122e.

Example 2:  Place the Zero flag into the Query flag, if Query is true.
Solution:
1. The trap bit is set to 0. The instruction bits are set to $001.
2. The instruction is conditional, so carry is set to 1.
3. The destination is r7, so DDD becomes $111.
4. The source value is in r7, so bit 7 is set to 0 and AAA become $111.
5. The bit to select is (decimal) 1, so bit 7 is 1 and BBB become $001.
6. The machinecode instruction is thus: $0001 1111 0111 1001, or
   decimal 8,057 or 0x1f79.
```

Instruction 2: Add

'Add' places the addition of two values into a given register.

Bit No.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	0	0	1	0	C	D	D	D	R	A	A	A	R	B	B	B

The instruction adds a value or register value to another value or register value, and places the result into the register given in the three DDD bits. If bit 11 is set to 1, the instruction is conditional. If bit 7 is set to 0, then AAA becomes the register from which to take the value. If set to 1, then AAA becomes a constant, from 0 to (decimal) 7. The same can be said for bit three and its relation to BBB.

If the result of the addition of the two values is larger than a 16-bit number, the carry flag is set (thereby providing a temporary 17th bit) and the remaining 16 bits hold the 16 least significant bits of the result.

```
Example 1:  Add one to the contents of r1 and place the result into r3.
Solution:
1. The trap bit is set to 0. The instruction bits are set to $010.
2. The instruction is not conditional, so carry is set to 0.
3. The destination is r3, so DDD becomes $011.
4. The first constant is r1, so bit 7 is set to 0 and AAA become $001.
5. The second constant is 1, so bit 7 is set to 1 and BBB become $001.
6. The machinecode instruction is thus: $0010 0011 0001 1001, or
   decimal 8,985 or 0x2319.

Example 2:  Multiply r5 by 2 and place the result into r0, only if the Query
            flag is 1.
Solution:
1. The trap bit is set to 0. The instruction bits are set to $010.
2. The instruction is conditional, so carry is set to 1.
3. The destination is r0, so DDD becomes $000.
4. The first constant is r5, so bit 7 is set to 0 and AAA become $101.
5. The second constant is also r5, so bit 7 is set to 0 & BBB become $101.
6. The machinecode instruction is thus: $0010 1000 0101 0101, or
   decimal 10,325 or 0x2855.
```

Instruction 3: Subtract

'Subtract' places the result of the second value minus the first into a given register. The rules are as with Add. If the result of the subtraction is zero or less, the Zero or Negative flag are set to 1 accordingly, and the value 'wraps around'

as in 2s-comp (See Chapter 3.7).

Bit No.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	0	0	1	1	C	D	D	D	R	A	A	A	R	B	B	B

```
Example:  Minus two from one and place the result into r4.
Solution: (NB. the result will be decimal 65535 and carry will be set.)
1. The trap bit is set to 0. The instruction bits are set to $011.
2. The instruction is not conditional, so carry is set to 0.
3. The destination is r4, so DDD becomes $100.
4. The first constant is (decimal) 2, so bit 7 is 1 and AAA become $010.
5. The second constant is 1, so bit 7 is set to 1 and BBB become $001.
6. The machinecode instruction is thus: $0011 0100 1010 1001, or
   decimal 13,481 or 0x34a9.
```

Instruction 4: Logical NAND

'NAND' works similarly to Add and Subtract in terms of its syntax:

Bit No.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	0	1	0	0	C	D	D	D	R	A	A	A	R	B	B	B

AAA and BBB can, once more, be registers or constants, the limits of the latter being similar to the other instructions as 7-bit values.

```
Example 1:  NAND r1 and r2 and place the result into r3.
Solution:
1. The trap bit is set to 0. The instruction bits are set to $100.
2. The instruction is not conditional, so carry is set to 0.
3. The destination is r3, so DDD becomes $011.
4. The first constant is in r1, so bit 7 is set to 0 and AAA become $001.
5. The second constant is r2, so bit 7 is 0 and BBB become $010.
6. The machinecode instruction is thus: $0100 0011 0001 0010, or
   decimal 17,170 or 0x4312.

Example 2:  NAND the value (decimal) 5 by itself and place the result
            into r0, only if query is set.
Solution:
1. The trap bit is set to 0. The instruction bits are set to $100.
2. The instruction is conditional, so carry is set to 1.
3. The destination is r0, so DDD becomes $000.
4. The first constant is (decimal) 5, so bit 7 is set to 1 and AAA = $101.
5. The second constant is the same, so bit 7 is 1 and BBB become $101.
6. The machinecode instruction is thus: $0100 1000 0101 0101, or
   decimal 18,517 or 0x4855.
```

Instruction 5: Rotate Left (ROL)

Bit No.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	0	1	0	1	C	D	D	D	R	A	A	A	R	B	B	B

```
Example 1:  Rotate r5 by 1 and place the result back into r5.
Solution:
1. The trap bit is set to 0. The instruction bits are set to $101.
2. The instruction is not conditional, so carry is set to 0.
3. The destination is r5, so DDD becomes $101.
4. The first constant is in r5, so bit 7 is set to 0 and AAA become $101.
5. The second constant is 1, so bit 7 is 1 and BBB become $001.
6. The machinecode instruction is thus: $0101 0101 0101 1001, or
   decimal 21,849 or 0x5559.

Example 2:  Rotate r1 by the value in r2 and place the result into r3,
            only if the query flag is set.
Solution:
1. The trap bit is set to 0. The instruction bits are set to $101.
2. The instruction is conditional, so carry is set to 1.
3. The destination is r3, so DDD becomes $011.
4. The first constant is in r1, so bit 7 is set to 0 and AAA become $001.
5. The second constant is in r2, so bit 7 is also 0 and BBB become $010.
6. The machinecode instruction is thus: $0101 1011 0001 0010, or
   decimal 23,314 or 0x5b12.
```

Instruction 6: Fetch from Memory

Our first memory instruction places the contents of a given data memory address into a given register.

Bit No.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	0	1	1	0	C	D	D	D	R	A	A	A	R	B	B	B

The setup is identical to add and subtract. The destination must be one of the registers, and the address must be given as two values, which are added to find the appropriate data memory address from which to get the value. Either AAA or BBB can be either a register or value, but the latter can only be 3 bits long. Therefore, using only constants, addresses 0 to (decimal) 14 can be accessed.

To get at higher addresses, you must go through registers. To access a 16- bit address (i.e. up to (decimal) 65,535), use a register and a value (such as r0+0, or 0+r4). If two registers are used together, their 32-bit value (rAAA+rBBB) is used. This is useful for JTT memory sizes larger than 16 bits (see MemSize, Chapter 6.2).

```
Example 1:  Place contents of memory address (decimal) 13 into r0.
Solution:   (N.B. There is more than one combination of constants to use.)
1. The trap bit is set to 0. The instruction bits are set to $110.
2. The instruction is not conditional, so carry is set to 0.
3. The destination is r0, so DDD becomes $000.
4. The first constant is (decimal) 7, so bit 7 is set to 1 and AAA = $111.
5. The second constant is (decimal) 6, so bit 7 is 1 and BBB = $110.
6. The machinecode instruction is thus: $0110 0000 1111 1110, or
   decimal 24,830 or 0x60fe.

Example 2:  Place contents of the memory address designated by the
            32-bit value r0+r1, placing result in r2, only if query is set.
Solution:
1. The trap bit is set to 0. The instruction bits are set to $110.
2. The instruction is conditional, so carry is set to 1.
3. The destination is r2, so DDD becomes $010.
4. The first constant is in r0, so bit 7 is set to 0 and AAA become $000.
5. The second constant is r1, so bit 7 is 0 and BBB become $001.
6. The machinecode instruction is thus: $0110 1010 0000 0001, or
   decimal 27,137 or 0x6a01.
```

Instruction 7: Store to Memory

The second memory instruction places the contents of a given register into the given data memory address. It is slightly different to the rest. Whilst the other instructions hold (in machinecode binary, from right to left) the source

and then the destination, Store to Memory holds the destination memory address followed by the source register or value. Thus, in the table below, S denotes the source constant or register.

Bit No.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	0	1	1	1	C	S	S	S	R	A	A	A	R	B	B	B

One other thing to note is that the source (bits 8-10) can only refer to a register, since there is not enough room for the extra bit needed to store whether the source is a register or constant value. The instruction otherwise works as the reverse of Instruction 6. To illustrate, the two examples below are similarly contrary to those of 'Fetch from Memory'.

```
Example 1:  Place contents of r0 into memory address (decimal) 13.
Solution:   (N.B. There is more than one combination of constants to use.)
1. The trap bit is set to 0. The instruction bits are set to $111.
2. The instruction is not conditional, so carry is set to 0.
3. The source is r0, so SSS becomes $000.
4. The first constant is (decimal) 7, so bit 7 is set to 1 and AAA = $111.
5. The second constant is (decimal) 6, so bit 7 is 1 and BBB = $110.
6. The machinecode instruction is thus: $0111 0000 1111 1110, or
   decimal 28,926 or 0x70fe.

Example 2:  Place the contents of r2 into the memory address designated by
            the 32-bit value r0+r1, only query is set.
Solution:
1. The trap bit is set to 0. The instruction bits are set to $111.
2. The instruction is conditional, so carry is set to 1.
3. The source is r2, so SSS becomes $010.
4. The first constant is in r0, so bit 7 is set to 0 and AAA become $000.
5. The second constant is r1, so bit 7 is 0 and BBB become $001.
6. The machinecode instruction is thus: $0111 1010 0000 0001, or
   decimal 31,233 or 0x7a01.
```

4.5 Mnemonic Exceptions and Keywords

There are a few exceptions and additions to the mnemonic syntax that do not appear in JTT's machinecode language. There are still strictly only eight JTT machinecode instructions, and there can only therefore be eight mnemonic instructions. Those below, however are neither instructions nor CLI or GUI commands.

The MaxMem Constant

The *MaxMem* keyword can be used in the place of the largest available memory address. Whilst JTT's actual memory size is set at default to 4,096, this value can be altered by the user via the MemSize command (see Chapter 6.2). As a result, whenever the memory size is changed, a program, which addresses, uses or exceeds the maximum memory address has to be altered to compensate.

In mnemonic commands, this can be taken into account by using the *MaxMem*keyword in any operations regarding the largest memory address. Then, when the program is compiled, the keyword is replaced with the necessary value for you. However, it should not be applied to machinecode.

```
Example:  Say a program needs to put the value decimal 255 or
          $1111 1111 or 0xff into the last, and the second-to-last,
          data memory addresses. The code would look something
          like this:

          r0=MaxMem;
          memory[r0+0]=$11111111;
          memory[r0-1]=$11111111;
```

The following code is also acceptable, and more efficient after compilation:

```
memory[MaxMem+0]=$11111111;
memory[MaxMem-1]=$11111111;
```

Please Note
The phrase 'MaxMem+1' is accepted, since it is syntactically correct. However, take care against this, since it always exceeds the maximum memory address.
The *MaxMem* constant is not case sensitive.
Because the *MaxMem* value is treated as a constant, it can be used in conjunction with the Load Constant instruction.
See also the comparison instruction extension (Chapter 5.4), which is also a mnemonic command extension.

Comments

A comment is a string, found in mnemonic instruction listings, which is intended to be read by a programmer but not to be executed as an instruction. Such devices allow the programmer to place notes or reminders inside a program at particular places. There are two comment keywords, both taken from other programming languages. These are the ';' (semicolon) character and the '#' (hash) character. They can be used at any place on a line, and the number of spaces preceding it (whether after an instruction or on a new line) does not technically matter.

During compilation of a mnemonic program into machinecode, these symbols and the rest of the line after them are ignored. Therefore, they may be used at the start of a new line, or after an instruction at the end of an old one. It is conventional to use the hash character to dedicate an entire line to a comment, and the semicolon to add a comment after an instruction. An entire mnemonic instruction line dedicated to a comment is treated as a non- instruction (a *null* instruction - see below).

When such comments are entered, they are stripped away from the instruction and will not appear with the program listings. However, the comments will be stored in a special database of the CLI consisting of as many entries as there are memory addresses. Whenever you are about to overwrite an instruction memory address that you have marked with a comment, you will be informed of the comment and asked to confirm the overwrite of the address.

Alternatively, you can view any or all comments using the comment command (see Chapter 6.2).

```
Example:  The first of the following example lines of code adds a
          comment after the given instruction. The second
          dedicates an entire in line to a comment.

          r6=MaxMem;      Places largest mem address into r6
          r0=r1+r2;
          # The preceding line adds r1 and r2 --> r0
```

Null Instructions

When JTT is first loaded, there are no instructions in instruction memory and data memory is filled with a series of 0s. This also occurs when the programmer uses the command in the CLI (see the Data command, Chapter 6.2). This way, when a programmer fills memory with instructions but leaves gaps between code, unwanted instructions the programmer did not ask for are not carried out. Such a marker in instruction memory, which tells JTT to carry out no action is called a non-instruction, or *null* instruction.

The JTT machinecode for a *null* instruction is 65534, or $1111 1111 1111 1110, or 0xfffe, and is therefore a trap instruction (see Section 4.4). However, in both mnemonic and machinecode listings, it is shown simply as the word *null*.

You can insert *null* instructions of your own by typing it as a mnemonic command, or using the above machinecode conversion.

Please Note
The *null* keyword is unique to this usage within JTT, and should not be used as (and indeed is not) a value in itself.

HALT Instructions

HALT is the only other instruction that makes use of the TRAP extension. Its machinecode form is 65535, or $1111 1111 1111 1111, or 0xffff. It causes the program to stop running permanently, and you will be returned to the CLI/GUI window from which you ran it in the first place.

Please Note

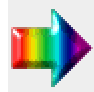
Because it uses all 16 bits, there is no physical room to store whether the *HALT* instruction is conditional, and so can only make use of the Query flag indirectly. To do this, you should use a conditional statement that points r6 to a *HALT* command elsewhere. *HALT* itself should be placed with care.

4.6 Helpful Tips: Basic

- Some instructions cannot be applied to some contexts. For example, 'Load 'Constant cannot load a value directly into a data memory address. To get around this, use the instruction on a register, then store the contents of the register in a memory address using the 'Store to Memory' instruction.
- Place a semicolon after every instruction entered. This is good practice for programming languages like C and Java, which utilise the same convention.
- If you do not currently wish to use a particular instruction or group of instructions, but may do in the future, 'comment out' the offending lines by prefixing them with the hash character (see Section 4.4).

4.7 Helpful Tips: More Advanced

- It is perhaps useful to know that the instruction r0=r0 is, in fact, decimal 0 in machinecode. Thus you may use it as a short cut to typing *null*. The latter is shown to avoid 'clogging up' the display, although the former is strictly true.
- Jump to other places in instruction memory by assigning new values to the Instruction Pointer (Register 6) via 'Load Constant'. For example, to move to instruction memory address (decimal) 20, use something like r6=0xf4. The registers are covered in more detail in Chapter 5.2.
- It might pay to 'modularise' large programs - i.e. split them throughout instruction memory into separate functions, or modules, and work on each separately. Leave large gaps between memory addresses, depending on the size of the overall program and the amount of work/expansion to be done to each, between each. When one module ends, 'jump' (see above bullet-point) to the next.
- To create loops (where the same sequence of instructions is repeated over until a particular condition is satisfied), use 'jump' at the end of the sequence twice - one to repeat if not satisfied and one to break the loop if otherwise. The 'q' (query) flag can be used here in conjunction with the '?' extension (see Chapter 5.4).



top | contents | homepage | e-mail your queries

5. Memory, Registers and Flags

In addition to the instructions, JTT obviously needs memory to place the data and instructions in. It also needs flags to check for certain conditions (commonly called 'if-statements') and registers in which to store data for manipulation, so that the resulting calculation can be put back in memory. All of these work (necessarily) in conjunction with instructions, and all will be used by the JTT programmer sooner or later.

5.1 Memory

Although memory has been touched on in preceding Chapters, there is more to say:

- Memory is a sequence of slots in which data is stored. The slots are called addresses, and each address has its own unique number.
- There are two memory types: instruction memory, which contains the list of instructions that make up a JTT program, and data memory, where the data used by the JTT program to be manipulated and calculated is stored.
- JTT has a default number of 4,096 memory addresses, numbered (decimal) 0 to (decimal) 4095, or $0 to $1111 1111 1111, or 0x0 to 0xfff. Thus it can store a maximum of 4,096 instructions and 4,096 pieces of data at any time.
- Any part of memory can be used in conjunction with any other. That is, an instruction anywhere in JTT's instruction memory can access data anywhere in JTT's data memory, as long as both exist at the same time whilst the program runs.
- Memory is accessed by the **Fetch from Memory** and **Store to Memory** instructions. A value can directly be stored in memory via **Load Constant**.

5.2 Registers

There are eight 16-bit registers, to be used and accessed directly by the programmer. Data cannot be manipulated directly in memory, as memory is simply just that - a place where data is remembered. So, for example, to add two data together, the two values should be fetched from memory and stored in two separate registers. Then, an instruction is carried out on these two registers, and the result placed back into a memory address. There are some other rules:

- Registers can also be used to store data temporarily - the less data is read from or saved to memory, the fewer instructions are likely to be needed and so the faster (and more efficient) the program is.
- The registers are addressed within a JTT program as 'r' followed by the register's number. There are eight of them, so the registers are called 'r0' through to 'r7'.
- There are two special flags, r6 and r7, which have their own

special rules and should not be used routinely to hold data. See Section 5.3 and Section 5.4 for more.

5.3 The Instruction Pointer (r6)

Register 6, the *Instruction Pointer* (IP), is updated by JTT automatically. It points to the instruction in instruction memory that is next to be carried out. Since all programs are carried out in sequence, the IP is incremented by one when the last instruction is dealt with. The IP can be changed freely by the user within a JTT program. This means that it is possible to 'jump' to another instruction in instruction memory, and carry out loops that keep running until a particular circumstance is reached.

- To jump to one instruction from another, address the IP as 'r6' using the **Load Constant** instruction, as normal.
- To carry out loops, use the *q*(uery) flag and jump forward to new code if the circumstance is reached, or backward to repeat the loop if not.

Details about both of these operations are given in Chapter 4.3 and 4.4. The query flag is dealt with in Section 5.4 (next).

Please Note
The IP is also commonly referred to as the "Program Counter" (PC) in standard low-level computer terminology.

5.4 Flags and the Status Word (r7)

There are eight information bits and eight flags. Each is held in the 16 bits of r7 - the *Status Word* (SW).

The leftmost 8 bits of the SW contain information that the user cannot access or change. This includes the current version code of JTT and the existence (or not) of additional utilities, such as the GUI. It is not necessary to know much about these - just be aware of their existence - and so they will be ignored from now on.

The rightmost 8 bits of the SW are flags. A flag is a single bit that represents a condition. If it set to 1, its condition was satisfied as a result of the last JTT instruction. If set to 0, it was not. Flags can thus be used (and indirectly be set) from within your JTT program to form what are known as '*conditional statements*'. Sometimes, you will not know what the result of an operation will be, and will want the program to act differently, depending on a number of possibilities. If the result of adding two numbers is zero, you may want it to do one thing. If not, you may want it to do something else.

The SW should be considered, and strictly is, read only. This means that you cannot place a value directly in the SW by using, for example, **Load Constant**. Instead, it is updated automatically as a result of the previous instruction that was carried out whilst running a JTT program, and the user can influence the status of any of the flags it holds. There is one exception to this rule - the *q* (query) flag.

Please Note
As with decimal, binary and hex values, the least significant figure is the rightmost figure, and the most significant figure the leftmost. As such, bit 0 is always the one on the right and bit 15 on the left, and should be read backwards from right to left.

Flags may be addressed by their number or name, and are detailed below. Afterwards, an explanation of each, and a description of their uses in JTT programs, are be given.

Bit No.	7	6	5	4	3	2	1	0
Name	e	m	l	v	c	n	z	q

Bit 0 the "Query Flag" [q], and the Comparison Instruction Extension

'Query' is possibly the most important flag of them all. It is the only flag that can be set directly by the programmer within JTT code, and the main way of using it is by using the BitSelect instruction. When a value is thrown at r7, only bit 0, the *Query* Flag, is affected. The rest of the flags are protected and tamperproof, and updated automatically as the program commences. As a result, BitSelect can be used to place the contents of one flag into the *Query Flag* for comparison.

From here, the *comparison instruction extension* (CE) can be used to determine the course of action. The CE is a question mark that is placed before an instruction. The instruction will then only be carried out if the Query Flag is set to 1. It will be ignored, and the IP will move on to the next line, if it is set to 0. For example, the JTT instruction '*r6=0xf*' will unconditionally jump to the instruction at memory address (decimal) 16. However, '? r6=0xf' will go there only if the *Query Flag* is set to 1.

Let us take a simple but typical example.

A program that adds two numbers together needs to jump to instruction memory address 0xf3 if the two numbers equal 50, or to 0xf4 if not. The syntax of the program will follow this pattern:

```
1. Add the two numbers together.
2. Minus 50 from the result made in Step 1.
3. The Zero Flag (see below) will be set to 1 if the subtraction in Step
   Two resulted in zero, or 0 if otherwise.
4. Place the contents of the Zero Flag into the Query Flag using BitSelect.
5. Using the '?' (CE) extension, tell the IP to jump to 0xf3 if the Query
   Flag is set, or 0xf4 if it is not.
```

Say value A is 10 and value B is 40. Taking into account the above steps:

```
1. 10+40=50
2. 50-50=0
3. Zero Flag is set to 1
4. Query Flag is set to 1
5. CE jumps to 0xf3 because Query Flag is set to 1
```

Now let's say A is 5 and B is 7. Taking into account the above steps:

```
1. 5+7=12
2. 50-12=38
3. Zero Flag is set to 0
4. Query Flag is set to 0
5. CE jumps to 0xf4 because Query Flag is set to 0
```

This, incidentally, is not the full limit of the usage of the CE. It does not have to jump to a different instruction memory address under certain conditions. The CE can be used to place a value into a register, perform a memory operation or in conjunction with any of the eight JTT instructions.

The subsequent flags work off the same principle, and may be used and manipulated in much the same way.

Bit 1, the "Zero" Flag [z]

Zero is always set to 1 when the result of the last instruction was zero, or 0 if otherwise.

Bit 2, the Negative Flag [n]

Negative is always set to 1 if the last instruction determined a result less than 0, or 0 if otherwise.

Bit 3, the "Carry" Flag [c]

The Carry flag serves as an 'overflow' flag for non-2s-comp values. If the result of the last calculation was too great to fit into a 16-bit register, then this bit is set to 1 and the 16 least significant bits are stored in the register. Otherwise, it is set to 0. This way, it serves as a "17th bit". Note that using the C flag in this way is hazardous, since the result may have been more than 17 bits long.

Bit 4, the "Overflow" Flag [v]

The Overflow flag serves the same purpose as the Carry flag, except that it works in conjunction with 32-bit numbers in twos-complement.

Bit 5, the "Carry or Zero" Flag [l]

The l-flag may be considered the "Carry or Zero" flag. It is set if either the 'c' or the 'z' flag (or both) are set. This means that if the last command resulted in a calculation that was out of bounds or absolute zero, it is set to 1, or otherwise it is set to 0.

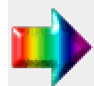
Bit 6, the "Negative and Zero" Flag [m]

The m-flag can be considered the "Negative and Zero" flag. It is used in twos-complement only, and is set to 1 if the last instruction resulted in zero, or to 0 if otherwise (i.e. even if the result was 0).

Bit 7, the "Any Zero" Flag [e]

The e-flag may be considered the "(Negative and Zero) or Zero" flag. Again used in twos-complement, it is set to 1 if either the M-flag is set or if the result was zero. In other words, it becomes 1 if the last instruction was negative-zero or positive-zero, or 0 if otherwise.

Please Note
With all flags, care should be taken to make sure whether the flag is set in 2s-comp or in normal 16-bit operations.
The "Any Zero" flag can be used in either mode, although its real benefit is in 2s-comp. Otherwise, it is identical to the Zero flag.
It is always good practice to make full use of every flag available, as long as the usual rules and protocols of use are acknowledged. However, you may wish to construct programs that emulate flags l, m and e via the other flags to get to grips with how they work.
Out of interest, you may be wondering why, given that JTT attempts to emulate a simple RISC instruction set, flags v, l, m and e exist, when just q, z, n and c should do. The reason is that JTT's architecture happens to support "physically" additional flags, and efficiency demands as little waste as possible.



top contents homepage e-mail your queries

6. The CLI Command Set

JTT sports a powerful CLI, or Command-Line Interface, which allows the user to do many things outside the emulator such as viewing/hiding the GUI, or viewing the online help database. These commands do not affect the content of JTT programs, or instruction and data memory: these are separate to the emulator. The CLI has its own memory, limited only to your computer's speed and capacity. As much as possible, characters are used onscreen to emulate graphics as if the user were navigating a GUI.

The CLI has many commands. However, due to the versatility of the command set, there are various points that should be considered:

1. Pressing *<ENTER>* with no command at a command-line will repeat the previous command (if any).
2. With some commands, options, or "arguments", can be given, allowing the user to customise what the command does when used. For example, the **h** command displays help on all commands. However, typing **h quit** displays information on quitting. Here, quit is the argument. Arguments are called "args" for short.
3. Sometimes, arguments are necessary for the command to run. Care should be taken to read the online help system (and this section) for each command before use.
4. Some commands can be called in more than one way. For example, the online help system can be called in three ways (see below).
5. Most commands can be called with a single letter. **quit** can be invoked by the letter **q**, for example. Remember that single-letter commands are case sensitive: lower-case **g** calls the command **go** whilst upper-case **G** toggles the GUI.

6.1 Command Syntax

The commands will be referenced, in this Manual and in JTT itself, using a specific format. The format is simple, and avoids some ambiguities that may crop up. Although similar to the instruction syntax covered in Chapter 4.2, there are many additions and reminders are always good. The syntax is shown in the table.

1. Necessary parts of a command are placed in pointed brackets (< and >). So, for example, the command **q** (quit) is referenced as <q>.
2. Optional parts of a command, i.e. arguments, are placed within square brackets ([and]). So, for example, the command **a** (assemble) can be followed by *from* and by a memory address, and is referenced as <a> [from x].
3. Required variables are denoted as x, y and z, as in the above example.
4. In circumstances in which one of a range of commands or options can be chosen, the choices are separated by |. So, since **quit** can be given as either **q** or **quit**, its syntax is <q|quit>. Alternatively, a shorter way of doing it (this

method of which is, incidentally, used more) is to write <q[uit]>.

5. All commands are case sensitive. Upper-case commands are considered different from lower-case. When in doubt as to which to use, use the online help system. As default, however, most commands are lower- case.

6.2 Command List

The complete list of commands, with usage guides, can be seen in the table below. Commands are placed in approximate order of likely common usage.

You may also use the following links to jump to the correct command:

about | assemble | data | debug | def; default | double list | double register | ed; edit | exit; quit | fr; file read | fw; file write | go | gui | help | lnum; line number | memsize | prefs; preferences | prompt | single list | single register | step | unassemble | user | quit; exit | ver; version | wipe

Please Note
The "larger" commands, needing greater elaboration, follow in the further sections.
In the following commands, the keywords from and to may be omitted where appropriate, as long as spaces are placed between commands and options.
In cases where instruction or data memory are accessed, such as in the go and assemble commands, the IP is changed to where information is to be placed. When you stop using the command, the IP will remain in its last place.
Press *ENTER* after every command. Only one command per line should be used.

Command	Usage Template	Example
?; h; help	<?|h[elp]> [topic]	help

The **help** command displays a complete list of all help topics, including acceptable command forms, template and usage information. If a topic is given as an option, help information regarding this topic only is given. If more than one topic fill the given criteria, or a few commands sound or are spelt similarly, all these will be shown.

As with requesters, Help has its own graphic characters to provide a cosmetic break and to break up the text. On Windows, box shapes will be drawn. However, on Unix machines, standard characters are substituted.

Command	Usage Template	Example
q; x; quit; exit	<q[uit]|x|exit> [topic]	x

The **quit** command exits JTT. If the current JTT session was not recently saved, the user is given the opportunity to save all work before exiting. If the resume command was set to true, all information in data and instruction memory, and all options, will be saved, to be resumed as you left off when you next load JTT.

Please Note
JTT can also be exited by pressing *CTRL* and *c* simultaneously. This acts as an *emergency-exit* function (e.g. if JTT's CLI locks up), which works at any time in the CLI. Be warned that, if used, all unsaved work will be lost, and the exit is immediate.

Command	Usage Template	Example
g; go	<g[o]> [from x [to y]]	go
		go from 10
		go from 0 to 0xf

The **go** command runs (carries out) the current program in instruction memory. If no args are given, the instructions are run from the current IP address. To run from elsewhere, type *from* and then the address to run from (x in the above template). The IP is then altered accordingly prior to running the first instruction (see *Example 2*).

To run a section of the total code, you can also stipulate a place in instruction memory at which to halt. This is useful for checking that a piece of code works without having to enter a temporary *HALT* command into your code. To do this, after typing the *from* keyword and address, type *to* followed by a second memory address. The second address will be the instruction carried out.

- With the exception of when the *to* keyword is used, the program will stop when a *HALT* command is reached. If the *HALT* instruction is not reached and instruction memory runs to the end, the IP will wrap around to memory address 0 and start from the beginning of the instruction memory.
- The user can a running program's progress by pressing **<ESC>**.

Command	Usage Template	Example
s; step	<s[tep]> [from x]	step
		step from 0

The **step** command runs a program in the same way as with the go command. However, this variant of the command is appropriate for debugging purposes.

After each command is executed, the JTT emulator will halt temporarily. Information will be displayed, such as the previous, current and next instruction, and the status of the registers (in particular the IP and SW). You will then have the chance of continuing to the next instruction or exiting step mode and returning to the CLI, to review the changes your program made to data memory.

If you just want to run the program but watch it each step it takes, use the step command and keep pressing *ENTER*.

Command	Usage Template	Example
G; gui	<G|gui>	gui

If the GUI is not currently open, then it will be opened. Otherwise, it will be hidden. Whilst the GUI is open, control will flow from the CLI to the GUI until the GUI is closed (by closing all of its windows). Hence, whilst the GUI is open, the CLI will not respond to input from the keyboard.

Please Note
Out of interest, you may have noticed this: If the CLI is locked when the GUI is open, you cannot type gui and close it via the CLI. The reason for this is that it may be made possible to use both the CLI and GUI simultaneously sometime in the future. For now, though, it's one or the other.
For more information concerning use of the GUI, see Chapter 7.

Command	Usage Template	Example
a; assemble; e; ed; edit	<a[ssemble]|e[d[it]]> [from x]	a from 0x15

The **assemble** mode (assembler) allows the user to type instructions (code) into the JTT emulator's instruction memory. Code will be accepted from the current IP address, or from memory address x if this argument is used.

Instructions are accepted in mnemonic or machinecode format. Mnemonics are assembled (compiled) automatically, and all instructions are parsed and checked for syntactic errors. If no errors are found, each instruction is placed into the appropriate instruction memory address and the assembler is ready to accept the next instruction to be placed into the next memory address.

If a memory address already contains an instruction, it will be overwritten without warning. The address that the current instruction will be placed into is shown in the assembler's instruction prompt, in hex notation. If the last memory address is reached, it will wrap around to memory address 0 and start from there.

To exit the assembler, press *ENTER* on a new, empty line. It will pass control straight back into CLI mode, where you can review and delete instructions added.

Please Note
To delete an instruction or series of instructions, see the **wipe** command (below).
To review your program(s), use the **step** command (above).
For tips with typing JTT code, read Chapter 4.4 and Chapter 4.5.

Command	Usage Template	Example
d; data	<d[ata]> [from x]	data from 64

The **data** command allows the user to type data (values) into the JTT emulator's data memory. If no args are given, data is entered from the current IP address. Otherwise, data is entered from the address given.

Rules apply as with assemble, regarding exiting data entry mode, validating data entries etc.. See also above notes. In data entry mode, values are accepted in decimal, binary and hex as long as the appropriate identifier (if any) is used.

Command	Usage Template	Example
u; unassemble	<u[nassemble]> [from x [to y]]	u from $10000000
		u from 0x0 to 0xff

The **unassemble** command is used to 'deconstruct' machinecode instructions from instruction memory and display them, in mnemonic format, in the CLI window. This command does not affect the instructions stored in memory; it merely lists the mnemonic representations onscreen.

If no args are given, this command lists the preset amount of instructions after the IP. If the from arg is given, it lists the preset amount from the given memory address. If both from and to args are given, it also lists, in batches of the preset amount, until the given end memory address is reached.

Please Note
The given amount is set to 25 at default, which is the maximum number of lines that can be shown in a standard MS-DOS prompt. However, this figure can be changed by use of the **lnum** command (see below).

Command	Usage Template	Example
l; list	<l[ist]> [from x [to y]]	list from 0x3f
		l from 0xf to 0x1f

This **list**s the contents of data memory from the current IP, or from data memory address x (to data memory address y) if given. The same rules for the display apply as with unassemble, except that data cannot be decompiled, and is instead displayed in decimal, binary and hex.

Command	Usage Template	Example
L; List	<L[ist]> [from x [to y]]	List from 0x3f
		List from 0xf to 0x1f

The **double-List** command works as with list above, except that adjacent memory addresses are paired into twos, and the corresponding 32-bit (16+16) value is shown. Such values are called *double* values, and more on using 32-bit numbers is given in Chapter 3.6.

Please Note
When using 32-bit numbers (as in **double-List**), you *must* use juxtaposed values starting with an even address. That is, an even address must be coupled with an odd address.

Command	Usage Template	Example
r; reg; register	<r[eg[ister]]> [reg]	r $100
		reg 7

The **register** command lists the contents of all registers, or, if a specific register is given as an argument, of a particular register. The argument's register number can be denoted in decimal, binary or even hex if being pedantic. r7 is given with its flag names, and both r6 and r7 are labelled with IP and SW, respectively.

Command	Usage Template	Example
R; Reg; Register	<R[eg[ister]]> [reg]	Register
		R $100

The **double-Register** command pairs up an even-numbered register and its succeeding register into twos, except for r6 and r7, which cannot be used in this way, and the corresponding 32-bit double value (*DWord*) is shown. Apart from these caveats, the rules of its use apply as with the single register command (above). For more information on doubles (DWords), see the double-List command (above) and Chapter 3.6.

Please Note
When using 32-bit numbers (as in **double-Reg**), you *must* use juxtaposed values starting with an even register. That is, an even register must be coupled with an odd register.

Command	Usage Template	Example
fr	<fr> <file> [to x]	fr example.jtt
		fr eg.jtt to 0x30

The **fr** (file read) command writes the sequence of machinecode instructions from the given file into instruction memory. The instructions are inserted into the current IP address. If the to option is used, the instructions are inserted at, and in sequence of memory addresses immediately after, the address given.

Such files are usually saved using the **fs** (file save) command (see below). Prior to loading, you will be told how many instructions are to be loaded in memory, and given a chance to confirm the action. If confirmed, the instructions will be written directly if there are no errors whilst loading file. All previous instructions in these spaces will be overwritten. However, memory addresses that are not needed to write the new information will be preserved.

Please Note

You can create your own '.jtt' files manually, but this is not recommended for novice or intermediate users and those who do not know how to write directly in machinecode. However, special files can be created in this way that are treated in special ways. For more on this, see Section 6.7.

Command	Usage Template	Example
fw	<fw> <file> [from x [to y]]	fw eg.jtt from 0x30
		fw eg.jtt from 0 to 0xff

The **fw** (file write) command writes the sequence of machinecode instructions from instruction memory into the given file name.

- If no args are given, the instructions are saved from the current IP address.
- If only the from arg is given, all instructions are saved from the given memory address and thereafter.
- If the from and to args are both used, the sequence of instructions is saved from the address given after the from arg, up to and including the address given after the to arg.

If a file already exists that has the name you specified, you will be asked to confirm overwriting that file. When loading the file back into memory, only those instruction memory addresses that are needed to fit the whole file will be overwritten. See also the notes for the fr (file read) command.

Command	Usage Template	Example
prefs	<prefs> [<s[ave]>|<l[oad]>|<d[efault]>]	prefs
		prefs s
		prefs default

The **prefs** (preferences) command allows you to load or save prefs, default to the original settings or show the current settings. Preferences are the settings that JTT is currently configured to use via other commands, such as **lnum** (see below).

- If no args are given, the current preference settings are shown, in a list. If they are the default settings (i.e. the prefs file does not exist), you will also be told.
- If you choose the save arg, the current settings will be saved in the prefs file. Whenever JTT is run, if this file is found, the settings in it are automatically used.
- If you choose the load arg, the last saved settings will be loaded and used. This allows you to revert to the settings you used when you first ran JTT this session.
- If you choose the default arg, the settings will be reset to the defaults (i.e. those that were in use when you ran JTT for the first time).

Please Note
The prefs file is called "prefs.jtt" and is located in JTT's home directory. It should not be moved or edited manually.
If you wish to use the default settings, rather than your own customised ones, each time you run JTT, select the default prefs and then save them (using the prefs command). Confirm your action at the dialogue. The prefs file will then be deleted.

Command	Usage Template	Example
w; wi; wd	<w[i|d]> [from x [to y]]	w
		wi from $00101101

		wd from 0xf0 to 0xff

The **w** (wipe) command has three functions. In any of its variants, all of the memory to which they apply will be erased if no args are given. If just the from arg is specified, memory is wiped from the given memory address until the end. If both the from and to args are given, memory is wiped from the given address up to and including the address given at the second arg.

- The **w** command wipes the contents of both data and instruction memory (from (and up to and including) the addresses given as args, if any).
- The **wi** command wipes the contents of just instruction memory (from (and up to and including) the addresses given as args, if any).
- The **wd** command wipes the contents of just data memory (from (and up to and including) the addresses given as args, if any).

Please Note
Unlike some other commands that take into account the IP when no args are used, the wipe command is total. If no args are used to specify from where to erase the contents of memory, ALL of its contents will be erased, not just those from the IP.

Command	Usage Template	Example
def; default	<def[ault]>	def

This command does very similar to the wipe and prefs commands, only is more powerful. After confirming the action:

- JTT's memory and registers are wiped
- The prefs settings reset to their saved settings (the prefs file)
- The GUI is closed if open
- The welcome/introduction screen is shown, as if JTT was just loaded
- JTT's memory size is reset to 4,096

Please Note
Any unsaved work will be lost after confirming the default action.
The default action cannot be carried out by beginner users. User level settings are changed via the **user** command.

Command	Usage Template	Example
lnum	<lnum> [value]	lnum 24

On any CLI display, there is a limit to the amount of lines of text that can be displayed onscreen at any one time. This limit is usually very limited, and specifying the help command without args would likely result in streams of text crossing the screen and disappearing without you having time to read it.

To solve this, the **lnum** (line number) command allows the user to specify a given number of lines that are to be shown at any one time. After this number, the display halts until you press *ENTER*, after which the next amount of lines are displayed, and so on until all the information has been shown. This way, text is shown in 'pages' at a time.

Typing the command with no args shows the current setting. With args, the user can specify (in decimal, binary or hex) a new value.

Please Note
The default number of lines (i.e. that which is the maximum shown at one time on a standard MS-DOS screen within Windows) is 25.
To display part of the previous page with the current one, select a lower value.

Command	Usage Template	Example
memsize	<memsize> [value]	memsize
		memsize 0xffff

At default, the memory size is set to 4,096. This means that both data memory and instruction memory can each hold 4,096 data and instructions respectively. However, this can be changed by the user.

Without args, the current setting is given. If a value is given (in decimal, binary or hex), this value is changed. Beware that the greater the value, the less physical memory will be available to your computer. You are required to restart JTT before the new setting can be used, due to the way JTT's memory is permanently set up when it is first loaded.

Acceptable values are powers of two. The minimum is (decimal) 63 (binary $0011 1111 or hex 0x7f), and lower values are recommended for those with those with a very small amount of physical memory available (usually 8MB or less and many applications running at the same time as JTT). The maximum is 2,147,483,647 ($1111 1111 1111 1111 1111 1111 1111 1111, or 0xffff ffff), and larger values are only recommended for those with a lot of physical memory available (usually 128MB or more and little or no memory-intensive applications running).

The 32-bit figure, is extremely extravagant and not recommended. It is available because JTT's architecture happens to support it as a theoretical maximum. However, this figure can become somewhat confusing and a maximum 16- bit figure is recommended.

For a guide as to how far you can push the upper limit on your system, increment the figure gradually (step by step) towards your target size and monitor the amount of memory available to your system using an independent application. If the figure drops below the 4MB mark, you should not increment any further.

Please Note
When testing larger memory sizes, make a note of the greatest successful amount you have previously used on this system for later reference. JTT should abort if there is not enough memory.
This command can only be used when the user level is set to *expert* (via the **user** command).

Command	Usage Template	Example
prompt	<prompt> [<["]string["]>|<t[ime]>|<d[ate]>]	prompt "JTT: "
		prompt time
		prompt d

The **prompt** is the string of text that appears before the flashing cursor, which tells you that JTT is ready for you to enter text. However, this prompt can be customised to display more useful information at the same time:

- Typing prompt with no args resets the prompt to its default setting.
- If a string of text (any text given within speech marks) is used as an arg, the prompt will be changed to the given text. This is useful if you need to remind yourself of something later, but should be kept short (a maximum of 10-15 characters is recommended).
- If or is given as an arg, the time or date that was current when the most current prompt was first shown is displayed.

Please Note
The time and date are taken from the system's settings, which may be wrong. Check your computer's time and date (and adjust if necessary) before use.

These arguments are meant to be used as a guide to the current time and date only, and will become increasingly inaccurate after very long periods of time. To re-synchronise them with the system, reuse the **prompt** command as appropriate.

Command	Usage Template	Example
user	<user> [beginner|normal|expert]	N/A

user displays the current user level, if no args were specified. Otherwise the level can be set to low (beginner), medium (normal) or high (expert).

In *beginner* mode, the user is encouraged to stick to decimal values in inputs. Thus, binary and hex values will never be displayed, and the user will be warned if they are used. Additional useful information is also given in particular circumstances.

In *normal* mode, the standard output messages are displayed, and the user can use dec/bin/hex values, which are also displayed where appropriate.

In *expert* mode, little currently differs from *normal* mode. However, advanced tasks such as memory allocation (using the **memsize** command) are permitted, and some messages are suppressed so that the user is confronted with a minimum of unuseful messages and information.

Command	Usage Template	Example
about	<about>	N/A

The **about** command displays the information provided at start-up. This includes the memory size, the current JTT version and a warning if you cannot use the GUI.

Command	Usage Template	Example
debug	<debug> [on|off]	N/A

The **debug** command displays the current debug state if no args are selected. Otherwise, it is switched on or off accordingly.

The **debug** mode monitors for illegal activity throughout JTT outside the traditional error procedures, and can even switch off error checking if it is found to have obscure flaws that make it behave in inappropriate ways. Although this is very unlikely, **debug** should be left ON at all times until it decides to switch itself off, in which circumstance you will be informed of the change of mode.

Command	Usage Template	Example
v; ver; version	<v[er[sion]]>	ver

The **version** command displays various version information, including that of the current instance of JTT, the version of Java you are using, and your computer's configuration and operating system.

6.3 Dealing with Loading/Saving Error Messages

The following error messages occur as a result of failures to load or save information from or to your computer. They are provided below in a format that will allow you to find what's wrong, read all about it and try out solutions. Loading queries are dealt with first, followed by those for saving. If all fails below, you can let the author know using the details given in Chapter 10.

Problem	**When loading, I am informed that JTT was unable to locate the file.**

Explanation	*If there is a problem with the prefs file, it probably was not found. In this case, set the options that you want and use the prefs command to save them. For information regarding using the prefs command, see Section 6.2.* *With JTT instruction files, the only files that are accepted are those with the suffix '.jtt'. If you do not type this in with a file name, it will be appended before an attempt is made to load it. If no path is given for locate the file, it will be searched for in the order given in the table below, where 'JTT:' is JTT's home directory:* *a) JTT:files/.jtt* *b) JTT:files/* *c) JTT:.jtt* *d) JTT:* *e) C:/.jtt* *f) C:/* *If it is then found, you will be warned if it did not have the .jtt suffix and this will be added when the file is next saved. You will then be shown its path.*
Solutions	If the JTT instruction file is still not found: • Check that you spelt the file name correctly. • If you supplied a path, check that this is spelt correctly and that it exists. • Search for the file yourself. In Windows '95 or '98, use the **find** utility (press the *Windows* key and '*f*'). • File names are case sensitive. Check whether the file is entirely upper-case, lower-case or a mixture of the two.

Problem	**I am told that the file is corrupted, and it will not load.**
Explanation	*Either the file is corrupted on the OS level (i.e. the system itself cannot read the file), or there is a syntax error in the file that prevents it from being loaded by JTT. The latter usually occurs with JTT instruction files.*
Solutions	If this does not help, try the following: • If the system could not read the file, you should either use a program independent of JTT to recover it, or get rid of it and start again. If it was one of the files that were there upon installation, reinstall (see Chapter V). • If there was one syntax error within the file, then JTT may attempt to repair it. You will be informed of the offending line. Such errors cannot be the result of illegal instructions unless the file was tampered with manually. • If there were more than one syntax errors, the file will not be loaded at all. • The errors may result from editing files manually, and may be corrected in the same way. For details on this, see Chapter 6.7).

Problem	**I am told that the file is not a recognised format, and it will not load.**
Explanation	*Again, this is most likely due to manual editing of the file.*

Solutions	You will have to look for the offending file yourself: • If it is the prefs file, you are best quitting JTT, going into the home directory, deleting the file called prefs.jtt, reloading JTT and re-saving your prefs. • If it is an instructions file, the first line of the file is likely to have a spelling mistake. Reedit and correct the file (see Chapter 6.7) and try again.

Problem	**I am told that the file cannot be saved.**
Explanation	*The file cannot be saved to the device/path that you specified.*
Solutions	There are various methods of solving the problem. • Check the device (e.g. C:\) and path (e.g. java\jtt\) you gave is valid. • Check that you gave a valid name to the JTT file. Names can contain most 'normal' characters, but not any of the following: / \ ~ • Check that the device is not write-protected. If it is a floppy disk, move the black box on the corner of the disk so that it creates a see-through hole. • Check that the device is not full. If it is, you may have to remove files and/or directories from it and try again, or save to a different device.

6.4 Dealing with Other Error Messages

Various other error messages may appear in the CLI for reasons other than for file input/output. They appear as warnings and fatal errors.

Warnings

Warnings are usually the result of incorrectly set variables that JTT. In such cases, JTT resets the offending variable to its default setting, and this is what JTT is warning you about.

Recoverable Errors

These do not result in the termination of JTT. In the case of recoverable errors, you should attempt to solve the error using the information given in the warning message. Such errors include incorrectly typed commands in JTT's CLI. Failing this, no further action may be necessary.

Fatal Errors

These are serious warnings, after which JTT cannot continue in a stable way. You may want to reload JTT afterwards and hope it does not happen again.

If it does not, there may be an obscure error within JTT's program code that attacks rarely, or you may have done something illegal that has not been picked up. In this case, check that what you type is correct before pressing *ENTER*.

If it does happen again, then you should make steps to find out what it is. If you can solve it from this, from the information given before JTT exited or from reading this Manual, then you should continue as normal. If not, or the problem happens again, you should note all the details and the circumstances that arose immediately prior to the error, and see Chapter 10.

6.5 Error Recovery

When JTT exits unexpectedly, it attempts to 'dump' (i.e. save) the current state of data and instruction memory, the registers, flags and current settings, to a file called "jtt.rec", which is located in JTT's home directory. When JTT is reloaded and this file is found, these details are also loaded (i.e. recovered) and you should be able to resume exactly as you were before JTT last exited.

The error recovery procedure is invoked either if a fatal error occurs, or if the user exits via the quit command and positively confirms the prompt asking to save. The file is only saved if possible (i.e. there is enough space on the drive and it is not write protected). Otherwise, the user is warned and given a chance to abort the exit.

6.6 The JTT Beans

JTT Beans: What They Are and How They Work

A JTT Bean (JTTB) is a file similar to an instruction file (i.e. one with the suffix '.jtt'), and is characterised by the extension '.jtb'. They are usually stored in the files directory. JTTBs have a number of uses:

- To provide practical examples for topics in the GUI's Help Database
- To provide the user with programs to study for ideas and programming practice
- To provide the user with various 'plug-in' modules they can use in their own JTT code, so that it is not necessary to rewrite the same code each time.

The JTTBs provided upon installation are of typical examples of what are likely to be popular groups of instructions.

Divide: Explanation

The Divide JTTB takes two values from data memory, divides the second into the first, and gives the result and its remainder, or the error code if an exception occurred. Remember that, after the two values in data memory, three additional addresses immediately after these have to be supplied. Exceptions are:

- Result is infinity (division by zero) (exception code (decimal) 1)
- First value is smaller than the second (exception code (decimal) 2)
- The address pointing to the first value is illegal (see below)
- There are not enough memory spaces to use (see below)

Divide's mnemonic code is good for studying the use of loops, Register 6 and the comparison instruction extension (see Chapter 5.4).

Divide: Technical Details

1. The address given in Register 0 (r0) is read. If this address does not exist, the value 0 is placed into r0 and the program HALTs.
2. It is checked whether four consecutive addresses immediately after this are available. If not, the value 0 is placed into r0 and the program HALTs.
3. The value in data memory at address r0, and at the address immediately after it, is read. The first is taken as the value to divide into (the divisible), and the second the value to divide by (the divisor).
4. A loop divides the divisor into the divisible. The result is placed into the data memory address after the two values read in (i.e. memory address r0+2), and the remainder after that (i.e. memory address r0+3).
5. The 'exception code' address is designated to be at memory address r0+4. Any exception codes are placed in this address, or 0 if no exceptions occurred.

Multiply: Explanation

The Multiply JTTB takes two values from data memory and multiplies them together. Remember that, after the two values in data memory, two additional addresses immediately after these have to be supplied. Exceptions are:

- The result is too large to fit into memory (exception code (decimal) 1)
- The address pointing to the first value is illegal (see below)
- There are not enough memory spaces to use (see below)

Multiply's mnemonic code is good for the same reasons as for Divide.

Multiply: Technical Details

1. The address given in Register 0 (r0) is read. If this address does not exist, the value 0 is placed into r0 and the program HALTs.
2. It is checked whether three consecutive addresses immediately after this are available. If not, the value 0 is placed into r0 and the program HALTs.
3. The value in data memory at address r0, and at the address immediately after it, is read.
4. A loop multiplies the first value by the second. The result is placed into the data memory address after the two values read in (i.e. memory

```
   address r0+2).
5. The 'exception code' address is designated to be at memory address r0+3.
   Any exception codes are placed in this address, or 0 if no exceptions
   occurred.
```

Exchange Sort: Explanation

The Sorting JTTB takes a group of consecutive values from data memory, sorts them and places the values, in order, back into the same memory addresses. Exchange Sort is a simple (but not the most efficient) sorting algorithm commonly used routinely to sort data. For more on how the algorithm works, see the comments inside the JTTB.

Remember that one additional address immediately after the list of values has to be supplied. Exceptions are:

- The address of the first value is illegal (exception code (decimal) 1)
- The address of the last value is illegal (exception code (decimal) 2)
- The last value is before the first value (exception code (decimal) 3)
- There is only one value, which cannot be sorted (exception code (decimal) 4)
- The address for the exception code is illegal (exception code (decimal) 5)

Sort's mnemonic code is also good for studying the use of registers for holding temporary and program-long constants.

Exchange Sort: Technical Details

```
1. The address given in Register 0 (r0) is read. If illegal, the value 0 is
   put into r0 and the program HALTs. Otherwise, the address is used for the
   exception code.
2. The addresses given in r1 and r2 are read. If either is illegal, the
   appropriate exception code is given and the program HALTs.
3. If the second address is less than or equal to the first, the appropriate
   exception code is given and the program HALTs.
4. The data in consecutive data memory addresses from that given in r1 to r2
   are sorted, and the data is placed back into the same sequence of
   memory.
```

6.7 Manual File Manipulation

JTTBs, along with instruction files, can be manipulated manually. To edit, use a standard ASCII text editor. In Windows, this should be something like MS-DOS's edit, or in SunOS or AmigaDOS, the ed command. There is no facility (at least as yet) to edit files from within JTT's CLI or GUI.

Please Note
Novice users of JTT should never attempt to edit files manually.

Manually Editing JTT Instruction/Program Files

The first line of a JTT file should contain the text 'file', in lower-case. Thereafter, instructions are stored in machinecode format (to save space). Each instruction appears on a new line, to make the file easier for the programmer to read. The memory addresses themselves are not stored, since they are not needed. Instead, as many instructions will appear as were saved into the file, whether the whole contents of memory or a portion thereof.

If there were comments to any of the lines, they will appear in the following syntax: {instruction};[comment]. The structure, format and order of a JTT file is shown in the table below.

```
file
{machinecode-instruction};[comment]
{machinecode-instruction};[comment]
...
{machinecode-instruction};[comment]
```

Manually Editing JTTB Files

The structure of a JTTB is similar, with some notable differences and additional rules:

```
1. The first line should (only) contain the text jttb, in lower-case
```

2. The next lines introduce the JTTB
3. Instructions are saved in mnemonic format
4. Comments are more clearly marked and placed further to the left
5. Comments are provided to explain most or all lines of code
6. Information is given at the bottom of the file, regarding which registers and memory addresses are used for inputs, outputs, storage and calculating and storing the result. Such information is given in commented lines.

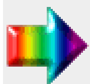
All rules except for 1 and 3 are conventional only, but should be adhered to when creating your own JTTB files. Note that the order is also important for comprehension purposes. Also feel free to leave spaces to break up the code to make it easier to read, but if this is done use comment symbols so that they are ignored.

The structure of a JTTB is once again fairly simple, and follows this basic format:

```
jttb
   ###################################################
   # {Name, Version Number, Author, Release Date}
   # {Comments on its purpose and usage}
   ###################################################
   {mnemonic-instruction>;          [comment]
   {mnemonic-instruction>;          [comment]
   ...
   {mnemonic-instruction>;          [comment]
   ###################################################
   # INPUTS:
   # {Registers}          {Input}
   # [Memory-Addresses}   {Input}
   #
   # OUTPUTS:
   # {Registers}          {Output}
   # {Memory-Addresses}   {Output}
   #
   # RESULTS:
   # {Registers}          {Result-or-Error-Number}
   # {Memory-Addresses}   {Result-or-Error-Number}
   ###################################################
```

6.8 Hints and Tips

- Make sure you adhere to the rules laid out when editing files manually, whether conventional or otherwise. A good way of doing this is to look at existing files and mimic their layout.



top contents homepage e-mail your queries

7. The GUI Command Set

One of JTT's strongest areas is its GUI. Whilst it has a powerful CLI and set of commands to go with it, it can be daunting to have to remember the format for using commands, and the protocols to follow when typing text. In this circumstance, a visual aid is always a benefit, in the same way that operating systems use GUIs to make it easier to use a computer.

JTT's GUI was programmed using recent techniques, meaning that some older versions of Java will be unable to run it. If this is so, you will be warned upon loading JTT and the GUI will refuse to open. You can still operate the CLI, however, which is just as useful. It is best in this case to skip Chapter 7 for now.

Otherwise, continue reading. The windows that make up the GUI are similar in basic functionality to those in most operating systems, and contain menus, sliders, arrows and gadgets to click on using the mouse. It is assumed that you have used such graphical interfaces before. Most of these will be explained briefly below, but if any of this is not even vaguely familiar, you should consult a manual for Windows, Unix, MacOS, Amiga Workbench or similar OS system GUI before reading on.

There are various parts of the GUI, or windows, that you can choose or choose not to view. Each has its own function, and is listed and outlined below.

The Main Window

This is the first (and by default, only) window that is shown when you open the GUI. It is a spreadsheet that allows to type in JTT instructions in mnemonic or machinecode format. Every memory address is listed horizontally in this window, and the hidden ones below (or above) can be accessed by using the slider or arrow gadgets to the right of the spreadsheet panel.

In each address shown, its instruction is given in mnemonic and machinecode. Additionally, the latter is given in decimal, binary and hex. If no instruction has yet been entered, the word null is shown.

JTT GUI: Main Window

File Window Help

address	mnemonic	dec	bin	hex
0	null		000	0x0000
1	null		000	0x0000
2	null		000	0x0000
3	null		000	0x0000
4	null		000	0x0000
5	null		000	0x0000
6	null		000	0x0000
7	null		000	0x0000
8	null		000	0x0000
9	null		000	0x0000
10	null		000	0x0000
11	null		000	0x0000
12	null		000	0x0000
13	null		000	0x0000
14	null	000000	$0000000000000000	0x0000
15	null	000000	$0000000000000000	0x0000
16	null	000000	$0000000000000000	0x0000
17	null	000000	$0000000000000000	0x0000

About JTT's Gui

Java Tiny Tim (Mk. II)

v0.1 rev 4.

(C) Copyright 2001 Michael Alcock et. al.

Gui created as from jtt v1.2.

Ok

Display information about the JTT GUI

To enter an instruction, click (which is always done with the left mouse- button) inside the square of the memory address to edit, whether it be mnemonic or machinecode. It should be highlighted with a thick box. Then click again, and you will see a black flashing cursor and the text that was already there will be highlighted. Type in the instruction, and press *ENTER*. If the instruction was invalid, you will be warned and its contents will be reset to its previous contents. Otherwise, the information will be entered, and all relevant boxes updated accordingly.

For information regarding the Main Window's menus, and opening the other windows detailed below, see Section 7.4.

The Register/Flag Window

This contains the contents of all eight registers (in decimal, binary and hex notation) as well as the status of each flag. The IP and SW flags are so identified.

The Data Memory Window

Whilst the Main Window concerns instruction memory, this displays the contents of data memory. It has a spreadsheet layout, and the data in each memory address is displayed in decimal, binary and hex. To edit values, see the Main Window above.

The Help Window

This is the window containing the help database. Full details are given in Section 7.5.

7.1 Dealing with Requesters and Error Messages

A requester (or dialogue) is a window that appears to inform you of a particular circumstance that has arose as a consequence of your actions. They serve the same purpose as error/warning messages in the CLI (see Section 6.4). However, there are, as you may expect, graphical differences. They contain a graphic, which shows you the status of the message, a message, and, below the text, gadgets that, when clicked, perform particular operations. Requesters come in various formats:

- **Warning requesters** (*graphic of exclamation mark in yellow triangle*) make you aware that a particular situation has arisen, for example when a command you typed into the Main Window was spelt incorrectly. These requesters only require you to click on the 'OK' gadget, and the program will continue as normal.
- **Alert requesters** (*graphic of exclamation mark in red circle*) ask you a question that should be answered before continuing. Usually, the gadgets on offer are labelled '**Yes**', '**No**' and '**Cancel**'. An example of an alert requester would be one asking you whether to save a program before overwriting it with a saved file.
- **Information requesters** (*exclamation mark in green circle*) are 'friendly' notices that inform you of non-imperative or immediate information.

Their use should otherwise be self-explanatory.

7.2 Using Pull-Down Menus

On many windows (and all of the JTT GUI's), there are titles that appear on a row at the top. Clicking on one of these titles reveals a smaller window that contains options pertaining to the title, such as help. These are pull- down menus.

To access a menu, click on its title. Whilst the mouse-button is held down, 'drag' the mouse-pointer down to the menu option that you want. If it is a sub-menu item, a further box will appear with more options. To choose an option, release the mouse-button.

7.3 Using Pop-Up Menus

This type of menu is less common. Similar to the pull-down menu, they appear when the right mouse-button is displayed. When this happens, hold down the mouse-button, 'drag' the mouse-pointer to your choice and let go to choose the option.

Pop-up menus appear on each of the JTT GUI's windows. They usually contain the main commands that can be accessed via the pull-down menus, but seldom contain all the available options. They therefore provide quicker shortcuts that allow you to get to the options provided for in their pull-down counterparts.

7.4 The Main Window Menu

On the Main Window, there are many menus that allow you to exit the GUI, load and save files, and display the various windows. As yet, these menus are uncomplicated and need not be explained here. You should get a feel for the menus by using them. The pop-up menu contains many of the same options, but give quicker access since there are less options to choose from. It is therefore likely to be easier to locate the option you want.

7.5 Help Database

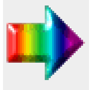
The Help Window is displayed via the Main Window's 'Window' menu title. Once open, there are two panels. On the left-hand side is a tree containing the main help topics. Click on one of the topics, and it will 'open out' to show its sub-topics. Click on one of these,

and it will be displayed on the right- hand side.

NOTE that due to major updates to the GUI, this part is subject to change soon. A preferences window is also planned.

7.6 Hints and Tips

None as yet.

 top contents homepage e-mail your queries

8. Other Sources of Starting and Further Information

It has been said from the earlier chapters of this Manual that the notations used within, and in JTT, do not necessarily reflect actual formal notations used in everyday programming languages and computer-related business. This is because JTT attempts to teach the underlying principles first, and some things are kept simpler for the sake of teaching them.

This means that JTT necessarily loses some of its technical accuracy. However, it attempts to keep as close as possible to accepted customs and practices, relating in the greatest way possible to the "outside world" it attempts to emulate.

For those of you who want to immerse themselves more shallowly at first, before attempting JTT, and for those who want a more in-depth look after reading this Manual, the following recommendations are given. All sources are freely available on the Internet. Those marked with (B) (basic) are introductory principles, whilst (A) (advanced) are principles which build upon those introduced by JTT. **Note** that this list is not exhaustive and, at this early stage in the compilation of this Manual, incomplete.

Binary

http://www.learnbinary.com/home.html

Twos-Complement (B)

http://www.artima.com/insidejvm/applets/InnerInt.html – an interesting program that demonstrates how 2s-comp is manipulated in binary etc.

9. The History of Java Tiny Tim

In the beginning, Drs. Barry Cook and Neil White, of Keele University, created Tiny Tim, a teaching processor designed to teach undergraduate students (primarily at Keele) the fundamentals of computer architecture. First released in 1998, it became integrated into the syllabus and has been ever since.

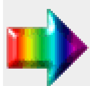
In September 2000, a third-year undergraduate, Michael Alcock, undertook the redesign and enhancement of the Tiny Tim program as a final-year project. This became known as "Java Tiny Tim". However, three of the main purposes of the project were to rewrite the package in Java, develop a GUI front-end and write the code as such that it was easily perusable to those interested in the design of large Java programs. These targets were largely met, and the report was written, detailing the reasons and methods involved in its design and construction.

However, after the project was submitted, Java Tiny Tim's author found that it lacked a comprehensive manual (the report being orientated towards its design and implementation rather than details of its use). Further to this, its code was much longer than necessary, the inefficiency of which was deliberate to make it easier to follow how the Java classes interacted with each other and worked.

The result was that the same author reinstated the project, with two main goals in mind: to write a comprehensive manual, and to optimise Java Tiny Tim's source code for speed and efficiency. The result was Java Tiny Tim Mark II, the package that accompanies this Manual. To avoid ambiguity, Tiny Tim Mark II was nicknamed "Big Tim", the second variation on the original theme. This Manual is also available in PDF format (albeit an obsolete and non-updated version), and on Keele University's Computer Science web site.

Big Tim was initially completed in mid November, 2001, as the result of many hours, much patience and an interest in Java programming.

In September 2001, another third-year finalist, Kerry _____, chose to enhance the package further and create a higher-level language based on the original Tiny Tim architecture, using Big Tim. This is due to be completed in July/August 2002. The Tiny Tim architecture is planned to be taught to Keele first-year undergraduates as from September, 2001.



10. Unsolved Errors, Queries and Bug Reports

First, make sure that you have exhausted all attempts to solve the problem from the information given by JTT and by this Manual, and that you have tried to solve it be reinstalling JTT from scratch. If files are corrupted as a result of manual editing, reinstall them. If they are not files provided upon installation, no action can be taken except for you to reedit and solve the problem yourself.

If there are problems that you have encountered whilst using JTT, which have remained unsolved after reading this Manual, then e-mail the following:

mike@cs.keele.ac.co.uk

with your query (note the '.co.uk' suffix, rather than '.com' which won't work). Make sure that you describe:

1.	A brief **summary** of the nature of the problem
2.	The **circumstances** under which the problem happens
3.	How many **times** it has happened (in a row or otherwise)**(1)**
4.	What you **typed** immediately prior to the problem**(2)**
5.	What text **display**ed as a result of the problem
6.	The **details** of the error message given by JTT (if any)
7.	Your computer's **configuration**, including the operating system and the OS version**(3)**
8.	The amount of total and used **memory** on your computer**(4)**
9.	The **configuration** of the version of Java you are using**(5)**
10.	The **JTT Version** that you are using (6)

(1)	*Details of circumstances in which JTT does work properly should also be included.*
(2)	*Include also where you typed it, e.g. in an MS-DOS prompt in Win98.*
(3)	*This information can be obtained in Windows by opening the Start Bar, then choosing **Settings** -> **Control Panel** -> **System** Icon.*
(4)	*This information can be obtained under System's **Performance** tab (see above).*
(5)	*This information can be obtained by typing "**java -v version**" (without quotes) in a CLI prompt (e.g. MS-DOS).*
(6)	*This is displayed when the JTT CLI first loads, and can also be viewed by typing "ver" at the command prompt.*

You should use the following template:

```
Problem:       JTT will not run under my system.
Circumstances: When I try to run JTT at a CLI prompt, I get an error message.
Frequency:     Happens every time without exception.
My input:      I type "java jtt" and the error occurs.
Details:       The error message says 'Exception in thread "main"
               java.lang.NoClassDefFoundError: jtt'
Configuration: Win98 2nd Ed. V4.10.222 A, GenuineIntel x86
Memory:        128MB (67% system resources free)
Java Version:  JRSE v1.3.1
JTT Version:   1.19 rev 155 (mk II)
```

The above example is short and can obviously be solved by choosing the correct JTT directory and/or installing the JTT package, but serves its purpose. In your query, please give as much detail as possible, and please, please make sure that you cannot solve it yourself first.

Replies are not guaranteed, but amendments to the Manual (and possibly to JTT) will be made as a result of problems with distinct and important merit. The author endeavours to reply to everyone as soon as possible, and (at time of writing) checks JTT's e-mail account around once a week.

Glossary

The glossary is currently empty.

top | contents | homepage | e-mail your queries

Index

Use the applet below to navigate through the indoglossary of terms, or use the search gadget to find what you're looking for. An explanation and link to the relevant page(s) of this Manual is usually provided. You can also use the arrow cursors on the keyboard and the Type Gadgets on the applet.

Applet and parameter code supplied by www.java.sun.com.
Modified 29/10/01 MRA.

;

Either your browser is not Java-enabled, or its Java capability has been disabled in the preferences or options settings.

Sorry, but for this reason you cannot use this page. Use Chapter 1 instead.