TASM Transputer Assembler User Guide

TASM Version 91.1
5/15/91

Copyright 1986-1991 by Logical Systems

Contents

1 Introduction
- Overview
- System Requirements

2 Usage
- Examples
- Option Information
- Option Descriptions

3 TASM Assembly Language Syntax & Semantics
- TASM Assembly Language Introduction
- TASM Pseudo-Opcodes
- Sample TASM Program
- Assembly Language Listing Format
- Assembly Language & Macros
- Operational Statistics
- Using the Preprocessor with TASM
- Notes on Using the Preprocessor

4 Appendix A: Error Messages
- Types of Error Messages
- Error Message Descriptions

5 Appendix B: Transputer Instruction Set
- Direct Functions
- Indirect Functions

6 Appendix C: TASM Internals
- Source Code Organization and Compiling

Introduction

Overview

TASM is a relocating assembler for INMOS Transputers. It supports standard INMOS mnemonics and allows splitting a program into separate pieces which are combined at linkage time. TASM is designed to be used in two ways:

1. As a post-pass to the TCX "C" compiler. The compiler generates an assembly language output file and TASM is used to turn it into relocatable format. The advantage of this scheme is the fact that the compiler can allow in-line assembly language without having to also have a redundant assembler built in.

2. As a stand-alone tool for doing assembly language programming on the Transputer. In this role it is often combined with the preprocessor from the "C" compiler (PP), which allows multi-line recursive macros, conditional assembly, include files, etc. TASM has been designed to be used with PP and can parse information PP provides to generate an assembly listing of source code which may have originated in many different files and been subsequently combined by PP.

The architecture of the Transputer requires that some of the code generation be delayed until the linker/locater stage to insure minimum length prefix strings are generated for all instructions. TASM supports this by determining which instructions can be "finished" and which cannot at assembly time. TASM finishes those which can be and provides information to the linker (TLNK), about the others.

TASM uses a multiple pass algorithm to determine which instructions can be "finished" and what the corresponding minimum length instruction prefix strings should be. The algorithm used doesn't guarantee minimum length prefixes in all cases (generating a minimum length program is a theoretically "hard" problem), but does a pretty fair job in a moderate amount of time.

System Requirements

TASM requires approximately 256K of program memory space to run. It should run in any environment which supports other major system development tools (compilers, etc). TASM does use a fair bit of disk space with temporary, output, and listing files. As an estimate, you should have disk space available that is twice the size of the input file for the temporary files, and space equal to the size of the input file for the output file (both types of files will be used at the same time when TASM is generating the output file). If you wish to generate a listing file you should have additional space available equal to twice the input file size. Note that a fair amount of I/O is done to the temporary files and they should be located on the fastest mass storage device available (see **Usage** section below for more information on how to specify this).

Usage

The general form of the TASM command line is:

```
tasm <input_filename> [<temp_directory>] [-[options]*]*
```

The basic idea is to specify the required input filename (complete with filename extension if not ".tal"), followed by an optional temporary file directory pathname, followed by any options needed. Note that in this case, since no explicit output filename has been specified, the output filename will default to that of the input filename, but with an extension of ".trl" in place of any extension the input filename had.

The temporary file pathname is used to tell TASM to use somewhere other than the current directory (or if the "TMP" environment variable exists, the directory it specifies), to hold the temporary files TASM generates. If possible, the temporary file pathname (whether explicit or via "TMP"), should be set to the fastest mass storage available (ideally a ram-disk). The process of assembling code for the Transputer may involve making the equivalent of many passes over the source text (most of which are done using temporary files).

Examples

Assume you wish to assemble a program stored on file "foo.tal"; The syntax needed is simply:

```
tasm foo
```

In this case TASM would use the "TMP" directory (or the current directory if "TMP" isn't defined), to hold its temporary files ("foo.1" and "foo.2"). If you had fast storage available on pathname "/fast" you could use:

```
tasm foo /fast
```

If you wanted to write the relocatable output to some file other than "foo.trl", say "foobar.huh", you would use the "-o" option flag followed by the desired output filename:

```
tasm foo /fast -o foobar.huh
```

Option Information

As seen above with the "-o" flag, an option flag may need a following parameter, although many option flags are simple switches which may be grouped together following a common "-" option flag lead-in. For instance, if you wish to toggle the verbose output mode and you wish to generate an assembly listing, the following command line will do it:

```
tasm foo -lv
```

Where the "l" indicates you want the listing and the "v" sets the output mode to verbose if TASM defaults to quiet and vice versa (actual default depends on the configuration of TASM). The above result could also be obtained by separating the option flags:

```
tasm foo -l -v
```

Please note, although in the above examples the option flags were in lowercase, uppercase is also allowed (some systems support nothing else)!

Option Descriptions

The following descriptions detail all the option flags available with TASM, what each does, and what additional parameters are required (if any):

Option flag: `-c`

This option is provided to "compress" the TASM output file. Doing this removes all the debug information. The big motivation is that this often cuts the output file size in half! It is particularly useful when building libraries or other chunks of code which aren't routinely debugged.

Option flag: `-l`

As mentioned previously, this option causes TASM to generate an assembly listing. The filename for the listing is the same as the output filename with an extension of ".lst". For example:

```
tasm foo -l
```

TASM would read the input file from "foo.tal", use "foo.trl" as the output file AND write the listing to "foo.lst". The "-l" option is not allowed if the original source text was not assembly language (see the "-t" option below). See later sections of this chapter for a description of the assembly language listing format.

Option flag: `-o <output_filename>`

This option flag allows you to explicitly specify the output filename (including extension), for TASM to use for the relocatable output file. If you don't provide an explicit extension TASM will use ".trl".

Option flag: `-q{0|1|2}`

These option flags allow you to control the level of prefix byte optimization that TASM will perform. The choices are "-q0" (no optimization, all references will be 8 bytes long), "-q1" (optimize to minimize the number of symbols and references which are passed on to the linkage phase), and the default, "-q2" (maximum optimization by deferring ALL boundary cases to link time). The "-q1" option is mainly provided for backward compatibility with pre-87.8 versions of TASM (where it was the default). The "-q0" option speeds up the execution of TASM in addition to potentially speeding up the execution of TLNK. During the early stages of program development use of the "-q0" flag on all files which make up a program (including those from libraries), will eliminate the otherwise required prefix optimization performed by TLNK and thus hasten the development cycle. On the down side the "-q0" option nearly doubles the program code size and execution time!

Option flag: `-t`

This option flag is used if the input file TASM is assembling was the result of a language translator. What this actually does is to inhibit TASM from counting input lines in an attempt to keep track of source input line numbers. TASM still accepts and updates its line number information in response to "#line" directives which are presumed to reflect the line numbers in the original source text. Using this option allows the line number information which is contained in the relocatable output file to represent the original source code line numbers instead of the (in this case), intermediate assembly language file line numbers. Using this option disables the generation of an assembly language source listing.

Option flag: `-v`

This option flag toggles TASM between the verbose and quiet output modes. Depending on the configuration of TASM this option will either cause additional information to be written to the user or disable same (the opposite of whatever the default setting is).

TASM Assembly Language Syntax and Semantics

The next several sections describe the syntax and semantics of the assembly language TASM accepts.

TASM has some syntax and semantic features in common with "C". Rather than repeat information which is familiar to many, we will refer you to a "C" reference manual for explanations about some features. The manual we recommend is:

"C" A Reference Manual
Samuel P. Harbison/Guy L. Steele Jr.
Prentice-Hall, Inc.
Englewood Cliffs, NJ 07632

Most other "C" texts also provide the level of description needed to understand the features TASM shares with "C".

TASM Assembly Language Introduction

TASM uses the standard INMOS abbreviations for instruction names (see appendix B for a listing of these). TASM is line oriented with one instruction allowed per line. Each line has the following format:

```
[<label_field>] [<opcode_field>] [<operand_field>]
```

Some sample assembly language statements:

```
test        j     @test ;Doesn't go anywhere very fast
            mint              ;Minimum integer instruction
            .db   "hello\n"   ;Define "C"-style string constant
label3                        ;Single label with no opcode
label4:                       ;Colons in labels are ignored
```

As you can see, comments are allowed after a trailing ";", and last until the end of the line. Comments may appear anywhere in a line (including the first column), but anything afterwards is ignored.

Labels are optional and must begin in the first column. The opcode field holds the instruction or pseudo-op name, it must not begin in the first column. The operand field contains any required parameters for the instruction or pseudo-op listed prior to it on the line. Fields should be separated with either spaces or tabs.

Labels and other TASM symbols are from 1 to 255 characters long. They are case sensitive. Labels begin with a letter, a "_" or a "?". They may contain those symbols plus digits. Labels may optionally be terminated with (or contain), one or more colon characters. Colon characters are allowed for compatibility with other assemblers and do not count as part of the label (you should not use a colon in any symbols you use in any operand fields).

The operand field follows the label field and contains either an instruction opcode or a pseudo-opcode. The pseudo-op's are begun with a ".", but are otherwise similar in form to opcodes (see the next section for information about them). The operand field contains different types of things depending on what the preceeding opcode or pseudo-op is. The types are:

• **"C" style constant expressions.** These include character constants and the other standard "C" features. You may also include symbols in constant expressions as long as they are defined in a ".set" pseudo-op prior to the constant expression in the file (no forward references allowed). Please see a "C" reference manual for a description of the constant expression syntax. Some examples:

```
'a'
-12
+1
(234+0x12)/022 + '\n'
'\033'+'\r'
23 ? 17 : 55
(help + me) / 0x3   ;'help' and 'me' must be
                    ; already ".set"
$12                 ;'$' causes wordlength scaling
```

In the above examples note that TASM allows a unary "+", which is not legal in "C" (pre-ANSI anyway). The value of a constant expression is just its numerical equivalent. Thus, a constant expression in a data definition pseudo-op just defines a byte or word location with the specified value. A constant expression as an operand to an instruction just uses the numerical value to compute a prefix string for the instruction.

Note that the '$' symbol may be used before the start of a constant expression to force the value of the expression to be scaled (divided), by the processor wordlength in bytes. Since the '$' may only appear as the first character in an operand field, it can only be used when the operand is a "pure" constant and not as part of the constant component of a non-constant expression. The expression to which the '$' operator is applied must be a multiple of the scaling wordlength or an error will be flagged.

• **"C" style string constants.** These include the normal character escapes allowed by "C" and are only legal for use with the define byte pseudo-ops (".db" or ".dbnz"). A string used with the ".db" pseudo-op will have the normal "C" style zero termination character while a string used with ".dbnz" will omit the terminator (the two pseudo-ops are otherwise identical). For example:

```
.db  "Testing 1 2 3\n"
```

• **Address expressions.** These consist of an symbol name followed by an optional constant expression. Some examples:

```
hello+27
frank
start -27+(0x66/2)
```

The value of the symbols used in the above examples is the address of the corresponding symbol definitions, NOT the relative offset from the current program counter to the symbol. These types of expressions are not fully bound at assembly time since the actual load address for the program is unknown. The term "bound" is used here to mean that the value of the prefix string for an instruction (or the value to store into a data word), can't be determined until the actual location for the symbol definition is assigned by the linker (TLNK). These expressions are allowed as operands of instructions or define word (".dw"), pseudo-ops (essentially word size "pointers").

• **Relative expressions.** These consist of a "@" followed by an optional symbol name, followed by an optional constant expression. These are allowed as operands of instructions or define word (".dw"), pseudo-ops. They allow the PC relative offset from the instruction (or data item), to the specified constant or symbolic address to be the value of the expression. Some examples:

```
@1
@hello +27
@ 100 - 0x10000
@ Gorge - ('a' + 'z') - 1
```

Note that there is a difference in value when a relative expression is used with an instruction versus its use in a ".dw" pseudo-op. When used with an instruction, the expression is evaluated so as to produce a prefix string which will correctly access the desired value (remember that the Transputer computes all relative offsets with reference to the memory location FOLLOWING the opcode byte of the instruction). In the case of the ".dw" pseudo-op, the value of the expression is the relative offset referenced to the START of the ".dw" memory location.

Note that if a relative expression contains a symbol it must immediately follow the "@". If the expression doesn't contain a symbol the constant expression is evaluated and the result is used as an address from which a relative offset is computed, starting at the "appropriate" current PC location, to determine the value to prefix the instruction with. For instance, if you wanted to create a jump to location 45 (decimal), you would use:

```
j     @45
```

Note that expressions which contain non-symbolic relative expressions can't be bound at assembly time since the load address for the instruction or data reference is unknown until link/locate time.

Some examples of relative expression instructions with symbol names:

```
cj   @hello + 10
call @Byte_output
.dw  @Beginning_of_data + 10
```

Symbolic relative expressions can be bound at assembly time (assuming the symbol is defined locally, and there are no unbound instructions which are between the symbol definition and the instruction or ".dw" which references it).

• **Difference expressions.** These consist of an optional wordlength scaling operator ('$'), a symbol name minus another symbol name, followed by an optional constant expression. Some examples:

```
@hello-@goodby + 1
hello-goodby
zip-zap + 21
$zip-zap + 4
```

To most assemblers this type of expression is just a normal "absolute" reference. TASM treats this as a special case since the variable length effects of the unbound prefix strings may cause this expression to be only partly bound at assembly time. This form of expression may be used with both instructions and ".dw" pseudo-ops (the '$' operator can only be used with instructions), but will probably be used most as an operand to "ldc" instructions which compute the branch length for "lend" instructions. For example:

```
begin                          ;Beginning of loop body
          <body of loop>
          <load pointer to "lend" parameter block>
          ldc  @end-@begin     ;Compute branch length for
                               ;"lend"
          lend                 ;Go back to beginning of loop
end                            ;End of loop body
```

Note that the optional '$' wordlength scaling operator causes the value of the remainder of the expression to be scaled by the wordlength (in bytes), of the processor family for which TASM is assembling code.

• **Floating point constants.** These are used with the ".real32" and ".real64" pseudo-ops to initialize memory locations with the equivalent number represented in either IEEE 32 bit or 64 bit binary format. The floating point constant syntax follows that of "C". TASM doesn't support floating point assembly time math, just the conversion operation (similar to initializing memory locations with the results of the "C" "atof" function). Some examples:

```
          .real32   0.0             ;Initialize a word to 0.0
          .real32   3.1415926,12    ;Initialize two words
          .real64   1.0,2.0,3e-39   ;Initialize three double
                                    ;words
```

TASM Pseudo-Opcodes

The previous section covered the operand fields of instructions and pseudo-op's in abstract, this section covers them in detail.

All TASM source files must begin with a pseudo-op which tells TASM what Transputer the code is being assembled for, since different versions support different instructions (and possibly different ways of generating code). The currently supported Transputers are the T2 series (T212/T222/T225), the T4 series (T400/T414/T425), and the T8 series (T800/T801/T805). To select a Transputer CPU type use one of the following pseudo-ops:

```
.all      ;Instructions for all 32 bit CPU types

.t212     ;T212/T222/T225 are described as "t212"

.t414     ;T400/T414/T425 are described as "t414"

.t800     ;T800/T801/T805 are described as "t800"
```

Note that ".all" is the default if no processor type is explicitly selected. The ".all" selection is primarily used when building code which is intended to run on any 32 bit Transputer (such as demonstration programs). Also note that both opcodes and pseudo-ops may be in either upper or lower case and that all pseudo-op names begin with a period.

All TASM source files should end with:

```
.end      ;No operand is required
```

This causes anything beyond it in the source code to be ignored. The use of this pseudo-op is not strictly required since TASM treats the end of the file as a defacto ".end", but it is important when TASM is being used with the preprocessor (PP). PP will otherwise remove any trailing comments and conditional assembly code from the input file to TASM, and thus remove the trailing stuff from any assembly listing which TASM makes.

Between these two pseudo-op's lies the body of the code. The remaining pseudo-ops are:

1. `#line <line_number> [<filename>]`

This violates the normal rules about pseudo-op's in that it begins with a "#", and it also starts in the first column. This is emitted by the preprocessor to update TASM about where the next input line to TASM really came from in the source file. The optional filename field indicates that the next line is also coming from a different original source file (the result of PP doing a #include). The information from these pseudo-ops is used to enable TASM to put the code from the original source file on the assembly listing, instead of the merged mess which PP generates. Use your "C" reference to find out further about this preprocessor directive.

---

2. `.align`

This pseudo-op tells TASM to word-align the next instruction or data statement.

---

3. `.db <value_for_byte> ["," <value_for_byte>]*`

This pseudo-op is used to initialize memory bytes to specific values. The "value_for_byte" field may be either a constant expression, or it may be a "C" style string (complete with automatic zero termination).

---

4. `.dbnz <value_for_byte> ["," <value_for_byte>]*`

This pseudo-op is identical to ".db" except the automatic zero byte termination of strings is eliminated. This pseudo-op was implemented to simplify the use of TASM with languages other that "C" (although the "C" string character escape sequences are still used).

---

5. `.ds <number_of_bytes>`

This pseudo-op reserves storage for the specified "number_of_bytes". Any constant expression may be used in the operand field. The space reserved in this way will be initialized to zero when the program is downloaded to the Transputer.

6. `.dw <value_for_word> ["," <value_for_word>]*`

This pseudo-op is used to initialize memory words (2 or 4 consecutive bytes depending on wordlength), to specific values. Note that this pseudo-op does NOT automatically perform word alignment; Use a ".align" prior to the ".dw" if alignment is necessary. The "value_for_word" field may contain the same types of operands as allowed for instructions (see the section on instruction operand fields). As mentioned in that secion, relative expressions applied to ".dw" are relative to the beginning of the word, NOT the location following, as is the case with relative references in instructions. You may not use "C" style string constants with the ".dw" pseudo-op.

---

7. `.emulate`

This pseudo-op enables instruction "emulation". This is used when you wish to simulate the effects of instructions which the currently selected Transputer processor type doesn't directly support. For example, you tell TASM you are using a T414 processor (via a ".t414"), then use a "DUP" instruction (after having given the ".emulate" directive). TASM will treat this as if you had given it an instruction of the form:

```
call @?DUP
```

Note that the instruction name called is always in upper case regardless of its original case in the input file. Also TASM will generate a ".ext" reference for symbols created this way if they haven't been previously encountered in the source file. It is up to the programmer to supply the simulation routine being called!

---

8. `.ext <symbol_name> ["," <symbol_name>]*`

The specified "symbol_name"s are declared to be defined "external" to this source file. It's presumed that the definitions will appear in other files which will be combined with the relocatable output of this one at link time. You may not both define a symbol within the current source file AND declare it ".ext". If a symbol which is declared ".ext" is not also referenced in the source file, the "external" reference is not included in the relocatable output file (no error is generated).

9. `.ldc <operand_field>`

This pseudo-op has the same syntax and semantics as the normal "ldc" instruction, but TASM and TLNK are free to use instructions other than "ldc" to load the desired value onto the top of the stack. This is useful when the immediate data to load is a large negative number and an equivalent code sequence of "mint"/"adc" can be used to load the same value in fewer bytes and instruction cycles. Another form of instruction sequence which may be generated is a "ldc"/"ldpi" sequence for cases when the current program counter is close to the desired address. This pseudo-op is extensively used by the TCX "C" compiler to minimize the length of static references. Note that this pseudo-op is affected by the ".rel" and ".norel" pseudo-op's when generating address expressions (code which computes the address of a symbol), for example:

```
.ldc zip + 21  ;Load address of "zip" + 21 bytes
```

Normally, this is allowed to use any of the optimization techniques to minimize the length of the generated code. However, if the ".rel" pseudo-op has been given, this form of reference is constrained to use the "ldc"/"ldpi" instruction sequence to maintain the runtime relocation capability. Note that all other forms of the ".ldc" instruction are unaffected by the current ".rel"/ ".norel" setting.

---

10. `.mod <module_number>`

TASM supports up to 256 different "modules", numbered 0 to 255. These modules are used to allow code and data which should be physically located in separate memory areas to be combined into the same source text stream. This facility corresponds to the "code" and "data" regions available with many assemblers, except 256 different modules are allowed. By default, if no ".mod" is given, the code and data which is present in the source file is placed into module 0. The linker (TLNK), allows you to select where each module from each source file will end up (or you may let it do the locating job for you).

---

11. `.noemulate`

This allows you to "turn-off" the instruction simulation facility which a previous ".emulate" enabled.

---

12. `.norel`

A complement to the ".rel" pseudo-op, this allows the address form of the ".ldc" instruction to generate the shortest/fastest possible code, ignoring the possibility of program runtime relocation. This pseudo-op is in effect by default and is used to "turn-off" the effects of a previous ".rel" pseudo-op.

13. `.pub <symbol_name> ["," <symbol_name>]*`

The specified "symbol_name"s are declared to be defined within this source file and are made "public", so that other files may refer to the symbol. You may not declare a symbol both ".pub" and ".ext". If a symbol which is declared ".pub" is not also defined in the source file, the "public" reference is not included in the relocatable output file (no error is generated).

---

14. `.real32 <fp_value_for_word> ["," <fp_value_for_word>]*`

This pseudo-op is used to initialize memory words (4 consecutive bytes), to values which correspond to the IEEE 32 bit floating point representation of the specified value. Note that this pseudo-op does NOT automatically perform word alignment; Use a ".align" prior to the ".real32" if alignment is necessary. The "fp_value_for_word" field may contain the same types of floating point constants that "C" allows.

---

15. `real64 <fp_value_for_double_word> ["," <fp_value_for_double_word>]*`

This pseudo-op is used to initialize memory double words (8 consecutive bytes), to values which correspond to the IEEE 64 bit floating point representation of the specified value. Note that this pseudo-op does NOT automatically perform word alignment; Use a ".align" prior to the ".real64" if alignment is necessary. The "fp_value_for_double_word" field may contain the same types of floating point constants that "C" allows.

---

16. `.rel`

Forces all ".ldc" pseudo-op's which follow to generate runtime relocatable code by using the "ldc"/"ldpi" instruction sequence for symbolic address expressions. This pseudo-op allows the generation of position independent code (assuming address expressions are not also used in initialized data areas, etc). Note that this pseudo-op will generally result in a somewhat larger/slower program since the other possibilities for ".ldc" instruction optimization are thereby disabled. This pseudo-op is NOT in effect initially in TASM and may be turned off once invoked by the later use of the ".norel" pseudo-op.

17. `.retf <workspace_adjust_constant>`

This pseudo-op is used by our "C" compiler as a function exit code short form. It translates into a "ajw" instruction with the specified "workspace_adjust_constant" used as the operand field, followed by a "ret". A couple of notes about the results of this pseudo-op: First if the constant value is zero, no "ajw" is generated. Second, any code after a ".retf" and before a "label" or pseudo-op is encountered is removed. Additionally, if the pseudo-op encountered is another ".retf", the SECOND ".retf" is also removed!

---

18. `.set <symbol_name> "," <constant_expression>`

The specified "symbol_name" is defined to have the value of the corresponding constant expression. This pseudo-op provides a "equate" capability for forward and backward references. The symbol name may be subsequently used in the "constant expression" part of the operand field for an instruction or pseudo-op which follows the ".set" in the source file. The symbol is otherwise treated identically to symbols defined as labels (it may be declared "public" for instance).

---

19. `.sym <symbol_name> ["," <address_expression>] ["," <constant_expression>]*`

This pseudo-op is used to hold debugging information for use by other tools in the Transputer Toolset. The arbitrary string, "symbol_name", is assigned a series of values including an optional, symbolic, "address_expression" (whose exact value is unknown until linkage time), and zero or more, 4 byte, "constant_expression" fields. The actual use of this statement involves a "convention" between the tool generating them (TCX), and a later debugging tool which interprets them. See the information about the "T_DEBUG_DATA", and "T_DEBUGSYM_DATA", relocatable records, in the "TASM/TLNK/TLIB RELOCATABLE RECORD AND FILE FORMAT" manual, for more information.

---

20. `.val <symbol_name> "," <constant_expression>`

This pseudo-op is similar to ".set" but is used for assigning purely local constant values to the "symbol_name". The references to a symbol defined this way must be strictly backwards and may not be external to the file. The primary advantage of ".val" over ".set" is that the symbol name used by ".val" may be redefined by a subsequent ".val" without having to create a new symbol. This is not possible with ".set" since both forwards and backwards references are allowed.

Assembly Language Listing Format

As mentioned elsewhere, TASM will generate an assembly language listing if the "-l" command line option flag is given. This file will be written on a filename which is the same as the input filename, but with an extension of ".lst" in place of any extension the input file had. TASM is designed to be used with the PP preprocessor, this carries over to the design of the listing facility for TASM. In particular, TASM can use information PP inserts in the input file to determine where the source text it is reading originally came from (say via "#include" PP directives). Using this information, TASM will find and use the original source code from wherever it came from when it creates the assembly listing.

TASM can't create an assembly listing if the original source code was written in some language other than assembly (see the "-t" option flag).

As a side note: If TASM detected assembly time errors it doesn't generate a relocatable output file. It also doesn't do the final "binding" passes it needs to resolve all the "relative" operand fields for instructions. This shows up on the assembly listing as instructions which are listed as "un-bound" (see below), when they really could have been bound.

The format of the assembly listing is:

```
<status><line><location><assembled_code><source_code>
```

The **"status**" field is used to show any error flags which were generated by that source code line, or a "." if something on the line was not completely bound at assembly time. The possible error flags are:

"D"	Duplicate symbol definition error.
"E"	Expression field error.
"F"	Floating point constant error.
"N"	Not implemented error (opcode/pseudo-op).
"O"	Opcode/pseudo-op unknown error.
"U"	Undefined symbol error.

See the corresponding error messages in appendix A for more information about what causes these errors. If the instruction was "bound", and didn't contain any errors, this field is blank.

The **"line**" field indicates which source code line this is. Note that the source code filename is shown on a banner at the top of the page initially, and a new page eject and banner is generated whenever the source code filename changes.

The **"location"** field. This indicates the current location counter relative to the currently active "module". This value will not be correct if the program contains any "un-bound" references, or errors, since the actual sizing and locating is delayed until linkage time. This field is shown for instructions or pseudo-ops which do anything "interesting", and unconditionally for the first line in a new source file.

The **"assembled_code"** field. This field contains up to the first 8 bytes of code the instruction or data pseudo-op generated. If the source line is "un-bound", and this information isn't known yet, this field is used to show the value of whatever the source code operand field contained in the form of a constant expression.

The **"source_code"** field. This contains the original source code as read from whatever file originally held it (assuming PP was used), or simply the TASM input file if PP isn't being used.

Assembly Language & Macros

When PP is being used with TASM, multi-line macros may cause many assembly language statements to be generated for a single "source" statement. This is handled on the assembly listing by simply showing the single original source code line. The problem is that the meanings of the various fields to the left of the source code line change somewhat. The basic rules are:

- The **"status"** field shows the first error encountered in the assembly statements which were generated by that source code line. If no errors need to be reported this field will contain a "." if any of the statements generated were unbound. If none of these conditions prevailed this field will be blank to indicate no trouble.

- The **"line"** field acts normally and shows the source text line number.

- The **"location"** field shows the location counter of the first instruction or pseudo-op in the macro expansion which generated any code. In other words, it reflects the start of the macro if anything "useful" happens.

- The **"assembled_code"** field shows the contents of the first operation in the macro expansion which placed anything in this field. It doesn't append the code generated by later instructions in the macro if the field isn't full yet.

- The **"source_code"** field acts normally and shows the original source text of the macro call.

Operational Statistics

Assuming no errors were encountered, TASM adds some operational information to the listing following the source code (this information is also written to standard output if you haven't disabled "verbose" output mode). The information written consists of the number of external symbols which were defined or referenced, the number of local symbols which were defined, and the number of local symbols which were "exported" in the relocatable output file for eventual binding by TLNK. The percentage of TASM's symbol table capacity which was used is also indicated. Note, within TASM, both local and external symbols use the same symbol table.

The last item on the listing (or standard output), is a count of the total errors encountered. This is a useful addition to the line-by-line error indications since multi-line macro expansion sometimes generates more than the one error which can be flagged on a given source line.

Using the Preprocessor with TASM

Using the preprocessor (PP), with TASM greatly improves the ease of programming, and the resulting readablity, of assembly language programs. If TASM is being used as a post-pass to the TCX "C" compiler, PP is not required (the "C" compiler handles those sorts of details with the help of PP itself). Assuming you are programming directly in assembly language, PP used with TASM offers the following improvements over using TASM by itself:

- **Macro processing.** PP allows both simple text replacement and powerful multi-line parameterized macros. Workspace offsets, symbolically defined configuration values, etc., are all good uses for this facility. The fancy parameterized macros are nice for creating in-line code, and PP has facilities for generating "unique" symbols which can be used to allow "local" labels and symbols within macro body expansions.

- **Include files.** PP allows nested include files to be used. This is useful when a set of configuration parameters is being shared by all the files in a program, but you only want to have one set of definitions.

- **Conditional Assembly.** Using PP allows you to do "C" style conditional assembly. This is useful when you wish to have two or more versions of a program share the same source text (and thus get updated together).

- **"C" style comments.** You may use "C" style comments in source code for TASM when you use PP (since it filters them out).

To find more out about these facilities consult your "C" reference manual. You may also want to consult "PP 'C' PREPROCESSOR USER GUIDE" for implementation-dependent information about PP.

Notes on Using the Preprocessor

The following example will preprocess and assemble a file named "test.pal":

```
tcc test.pal +a-l -c
```

Note that the "+a-l" directive tells TCC to pass a "-l" directive to TASM telling it to generate a "test.lst" assembly language listing file. The "-c" flag tells TCC that linking will not be necessary. The relocatable output file will be written on "test.trl".

A few notes should be mentioned about using PP with TASM:

1. If you are having trouble, or are unsure where a problem lies, check the output file written by PP to see what TASM is really getting as input. This is necessary since TASM shows the original source text on the assembly listing, not what it actually read as processed by PP. This is particularly useful in debugging macro's, since the assembly listing only shows the macro "call", not the subsequent expansion.

2. Within macros you should be careful about using ";" assembly language comments. Remember that these comments are NOT comments to PP and it will pass them on through to TASM. A place where this crops up is when you define a symbol to have some value in a "#define" macro and follow it with a ";" comment in the source text. The result is that anything you place in the operand field AFTER the spot where the macro replacement is done, gets commented out! As a general rule you should use "C" style comments for anything involving macros.

Appendix A: Error Messages

Types of Error Messages

There are three classes of error messages which TASM can generate:

- **Warnings**. These are used to report problems which aren't severe enough to cause TASM to abort (exit with a non-zero return value). These messages usually indicate trouble which isn't immediate, but may be soon! The format for warnings is:

```
WARNING: message_text
```

- **Non-fatal errors.** These are used for reporting actual error conditions which will affect the return value given when TASM exits. If one or more non-fatal errors are encountered TASM will return a non-zero return code, otherwise it will give a return code of zero. Another result of encountering non-fatal errors is that the generation of a relocatable output file is inhibited (although if a assembly listing was requested it will be generated). The format for non-fatal errors is:

```
<filename> @ line_number: message_text
```

Where the "<filename>" field indicates the current source code file being read, the "line_number" field gives the line where the problem was detected, and the "message_text field indicates the actual problem encountered. Note that non-fatal errors are also displayed on the listing (see the **Assembly Language Listing Format** section for a description of the format).

- **Fatal errors.** If the problem detected by TASM is so severe that it can't continue operating, it will give a "fatal" error message:

```
FATAL: message_text
```

After printing one of these messages, TASM will immediately exit with its error return code set (non-zero).

Error Message Descriptions

The following descriptions list the various error messages which TASM can generate (in alphabetic order):

```
<filename> @ line_number: Duplicate symbol definition:
symbol_name
```

The named symbol was either defined more than once, or defined once and mentioned in a ".ext" pseudo-op.

```
<filename> @ line_number: Expression field error
```

This error is generated whenever an illegal expression is present in the operand field of a opcode or pseudo-op. A few of the possible causes:

- Having an expression field which is not representable in 16 bits when assembling for a 16 bit processor.
- Using anything other than a string or a constant expression with a ".db" pseudo-op. If you want to reference an address you need to use a ".dw" instead.
- Using a "relative" reference within a ".dw" pseudo-op. Only constant expressions or address references are allowed there.
- Using a module number which is outside the range of 0 to 255 which is allowed for ".mod" declarations.

```
FATAL: Corrupted temp file: filename
```

This error usually occurs when the contents of a temporary file get corrupted by the file system somehow. If you have been changing TASM or recompiling it for another system, this error message indicates that the "type" field in one of the internal temporary file records was not one of the allowed types. This generally happens when you make a change to one of the places which adds or removes temporary file records without changing all the other occurences (you will generally need to make changes to files "tasm2.c", "tasm4.c", and "tasm5.c" together).

```
FATAL: Error reading input file: filename
```

TASM got an error return during one of its read operations on input file "filename". This usually indicates trouble with whatever mass storage device is being used, and/or a corrupted input file. If the preprocessor (PP), was used to prepare the input source file AND a listing is being requested, this error could indicate problems have cropped up in one of the source files between the time PP originally read it, and when TASM re-reads it to generate the assembly listing.

```
FATAL: Error reading temp file: filename
```

TASM got an error return during one of its read operations on temporary file "filename". This usually indicates trouble with whatever mass storage device is being used.

```
FATAL: Error setting stream buffer for file: filename
```

This error results when TASM is compiled with a non-zero IOBUFSIZE in file "taldef.h" but is unable to explicitly set the temporary file I/O buffer using "setvbuf" during execution. The return code from the "setvbuf" call is what actually triggers this error. As a workaround you can set IOBUFSIZE to 0 and recompile TASM, or you can figure out what is wrong with your "C" library. The file listed is the temporary file to which TASM was attempting to attach the buffer.

```
FATAL: Error writing listing file: filename
```

At some point TASM was unable to write to the named listing file. This generally occurs because of insufficient file space.

```
FATAL: Error writing output file: filename
```

TASM detected an error while it was writing the relocatable output file. This error generally occurs when insufficient disk space is available for the output file, as well as the temporary files which also exist during this period.

```
FATAL: Error writing temp file: filename
```

At some point TASM was unable to write to the named temporary file. This generally occurs because of insufficient space on whatever device the temporary files are being written on (either the "TMP" directory, the current directory, or a special "fast" one selected via the command line).

```
FATAL: Insufficient stream buffer memory for file: filename
```

If the value of IOBUFSIZE in "taldef.h" is non-zero, TASM will explicitly allocate temporary file I/O buffers (via "malloc" calls). If the memory can't be obtained for one of these buffers, this error message results. The filename listed is the one for which the buffer was intended. To get around this problem you should try to increase the amount of available "C" heap memory. If you are using TASM on a PC, get rid of any unnecessary memory resident programs. As a last ditch effort you can reduce the value of IOBUFSIZE and recompile TASM, but TASM execution speed will suffer noticeably.

```
FATAL: Insufficient symbol table string memory
```

TASM was unable to obtain (via "malloc" calls), enough memory to hold all of the symbols and labels used in the input file. The obvious solution is to reduce the number and length of the symbols in the input file. If you are using TASM on a PC you should try eliminating unnecessary memory resident programs as a first step in getting more memory.

```
FATAL: Line too long in input file: filename
```

TASM read an input line which was longer than 300 bytes (as the release version is configured). This error is generally the result of self-recursive macro expansion by the preprocessor (PP), or the use of a filter program on the input source file which removed the end-of-line markers.

```
FATAL: Output file name same as input
```

You have the same filename specified for both input and output. Remember that the default output filename extension is ".trl".

```
FATAL: Symbol table full
```

As configured in the release version, the symbol table can hold 4096 entries. This value may be increased if TASM is being run on a machine with a larger than 64K byte direct addressing range. Note that the symbol table size must be a power of two to make the hashing function work. If you can't increase the symbol table size you will have to break the input file up into separate pieces.

```
FATAL: TASM internal error #XXX
```

These errors should never occur! If one does it generally indicates a violation of one or more prefix optimization "constraints". If this error message does occur, please send a machine-readable copy of the offending TASM input file together with a description of what command line switches were used to either Logical Systems or the dealer where you purchased the product. Be sure to indicate what operating system TASM was running under and the complete text of the resulting error message (plus any other information you feel is pertinent). As a workaround, you can try adding, deleting or moving around bits of code in your program to see if you can avoid the exact sequence of optimization steps which provoked the problem.

```
FATAL: The size of SLONG is not correctly configured
```

This error message can only appear when you are recompiling TASM. It indicates that the "typedef" for SLONG which appears in "taldef.h" is set for a storage class which is less than 4 bytes long. The SLONG storage class MUST be signed for TASM to operate correctly.

```
FATAL: Unable to close input file: filename
```

You can only get this error message when you ask TASM to generate a listing file. It indicates that TASM was unable to close the named file during the process of re-reading whatever source files actually made up the input source file TASM read (assuming the preprocessor was involved), and generating the resulting listing.

```
FATAL: Unable to generate non-assembly language listing
```

This error is reported whenever both the "-l" and "-t" switches are given. You can only use one of these switches at a time.

```
FATAL: Unable to open input file: filename
```

The open attempt for the input "filename" failed. Verify that the input file exists and that the filename extension is correct (remember that ".tal" is the default if none is specified). If the preprocessor (PP), is being used with TASM, AND an assembly listing has been requested, this error message can also be generated. This occurs if one of the source files which PP used to create the input file for TASM was no longer there when TASM tried to re-read it to generate the listing. You can tell which of these two cases is the problem by noticing which input filename is mentioned in the error message.

`FATAL: Unable to open listing file: filename`

TASM was unable to open the listing "filename". The filename is created by taking the filename from the input file (and input file pathname), and appending the extension ".lst" in place of any extension the input file had.

`FATAL: Unable to open output file: filename`

TASM was unable to open the output "filename". This filename is either the default one generated using the input filename with a new extension (".trl"), or it was explicitly specified by you using a "-o" option flag.

`FATAL: Unable to open temporary file: filename`

The open attempt for the temporary "filename" failed. This filename includes whatever directory pathname was specified for temporary files.

`FATAL: Unexpected EOF in input file: filename`

This error is encountered when a listing is being generated and TASM is reading the various source files which the preprocessor (PP), used to create the input file. This error indicates that TASM found one of the input files was shorter than PP lead it to believe with information passed via "#line" statements. This error may also be generated without the help of PP, if the input source file has somehow gotten corrupted between the time TASM read the input code from it and when it was re-read to generate the listing file.

`<filename> @ line_number: Floating point constant error`

This error message is generated for floating point constants used with either ".real32" or ".real64" which are out of range of the particular IEEE format selected. This usually means an error in a mantissa or exponent field.

`<filename> @ line_number: Not implemented (pseudo-op)`

This error message is generated for opcodes or pseudo-op's which are not yet implemented, but whose names have been reserved.

```
<filename> @ line_number: Opcode/pseudo-op unknown: opcode_name
```

The named "opcode" appeared in the opcode field of an instruction but was not recognized by TASM. This is generally caused by not declaring what type of processor TASM is assembling for (".T414", etc.), or using an instruction which is not valid with the selected processor type.

```
WARNING: Unable to close output file: filename
```

During the cleanup process TASM removes the output file it creates if any errors were detected during operation. This error message indicates that TASM was unable to close the output file. Causes include the normal spectrum of file system related maladies.

```
WARNING: Unable to close temp file: filename
```

During the cleanup process, prior to TASM terminating, the temporary files are closed and deleted. This message indicates that TASM was unable to close the named temporary file (something is probably happening to the file system).

```
WARNING: Unable to remove output file: filename
```

During the cleanup process TASM removes the output file it creates if any errors were detected during operation. This error message indicates that TASM was unable to delete the output file. Causes include the normal spectrum of file system related maladies.

```
WARNING: Unable to remove temp file: filename
```

During the cleanup process, prior to TASM terminating, the temporary files are closed and deleted. This message indicates that TASM was unable to remove the named temporary file (something is probably happening to the file system).

```
<filename> @ line_number: Undefined symbol: symbol_name
```

This error message is generated when the named symbol is referenced but not defined within the input file (either by a label or ".ext" pseudo-op).

Appendix B: Transputer Instruction Set

The following descriptions of the Transputer instruction set are only intended for purposes of illustrating which instructions TASM can assemble. Please consult the appropriate INMOS documentation for information about instruction set formats and the internal architecture of the various CPU's.

Direct Functions

There are 16 direct functions, executed by all the INMOS Transputers, which can have operands. They are (in alphabetic order):

Instruction	Hex Value	CPU	Description
ADC	8	All	Add constant
AJW	B	All	Adjust workspace
CALL	9	All	Call subroutine
CJ	A	All	Conditional jump
EQC	C	All	Equals constant
J	0	All	Jump
LDC	4	All	Load constant
LDL	7	All	Load local
LDLP	1	All	Load local pointer
LDNL	3	All	Load non-local
LDNLP	5	All	Load non-local pointer
NFIX	6	All	Negative prefix
OPR	F	All	Operate (meta instruction)
PFIX	2	All	Prefix
STL	D	All	Store local
STNL	E	All	Store non-local

Indirect Functions

The use of the OPR instruction, in conjunction with the operand register, allows a large number of "indirect" instructions which are built using prefix strings to OPR. The following instruction list shows the indirect instructions, sorted in alphabetic order. Since INMOS makes more than one type of Transputer, the list has a "CPU" column which indicates whether the particular instruction is supported by at least some members of both the 16 and 32 bit Transputer families (listed as "16/32"), some non-16 bit (ie. 32 bit), processors ("All"), only by the 16 bit machines (T212/T222/T225, listed as "T212"), only by the non floating point 32 bitters (T400/T414/T425, listed as "T414"), or only by the floating point processors (T800/T801/T805, listed as "T800").

In addition, the floating point processors support a FPENTRY instruction which allows the current value in the A register to be used as an extended floating point operation code. TASM implements these extended operation codes as "macro" instructions which consist of a LDC with the appropriate extended code, followed by a

FPENTRY. These instructions are listed as "SEQ" (INMOS terminology), in the CPU column and are only available on the T8 processors.

Instruction	Hex Value	CPU	Description
ADD	05	16/32	Add
ALT	43	16/32	Alt start
ALTEND	45	16/32	Alt end
ALTWT	44	16/32	Alt wait
AND	46	16/32	Boolean AND
BCNT	34	16/32	Byte count
BITCNT	76	16/32	Count bits set in word
BITREVNBITS	78	16/32	Reverse bottom N bits in word
BITREVWORD	77	16/32	Reverse bits in word
BREAK	B1	16/32	Breakpoint
BSUB	02	16/32	Byte subscript
CCNT1	4D	16/32	Check count from 1
CFLERR	73	T414	Check real32 fp infinity or NAN
CLRHALTERR	57	16/32	Clear halt-on-error
CLRJ0BREAK	B2	16/32	Clear breakpoint flag
CRCBYTE	75	16/32	Calculate CRC on byte
CRCWORD	74	16/32	Calculate CRC on word
CSNGL	4C	16/32	Check single
CSUB0	13	16/32	Check subscript from 0
CWORD	56	16/32	Check word
DIFF	04	16/32	Difference
DISC	2F	16/32	Disable channel
DISS	30	16/32	Disable skip
DIST	2E	16/32	Disable timer
DIV	2C	16/32	Divide
DUP	5A	16/32	Duplicate top of stack
ENBC	48	16/32	Enable channel
ENBS	49	16/32	Enable skip
ENBT	47	16/32	Enable timer
ENDP	03	16/32	End process
FMUL	72	All	Fractional multiply
FPADD	87	T800	Floating point add
FPB32TOR64	9A	T800	Convert bit32 to real64
FPCHKERR	83	T800	Check floating error
FPDIV	8C	T800	Floating point divide
FPDUP	A3	T800	Floating point duplicate
FPENTRY	AB	T800	Floating point unit entry
FPEQ	95	T800	Floating point equality
FPGT	94	T800	Floating point greater-than
FPI32TOR32	96	T800	Convert int32 to real32
FPI32TOR64	98	T800	Convert int32 to real64
FPINT	A1	T800	Round fp to floating integer
FPLDNLADDDB	A6	T800	Floating ld non-local and add real64
FPLDNLADDSN	AA	T800	Floating ld non-local and add real32
FPLDNLDB	8A	T800	Floating load non-local real64
FPLDNLDBI	82	T800	Floating ld non-local indexed real64
FPLDNLSN	8E	T800	Floating load non-local real32
FPLDNLSNI	86	T800	Floating ld non-local indexed real32
FPLDNLMULSN	AC	T800	Floating ld non-local and mul real32
FPLDNLMULDB	A8	T800	Floating ld non-local and mul real64
FPLDZERODB	A0	T800	Floating point load zero real64

FPLDZEROSN	9F	T800	Floating point load zero real32
FPMUL	8B	T800	Floating point multiply
FPNAN	91	T800	Floating point test for NAN
FPNOTFINITE	93	T800	Floating pt test for NAN or infinite
FPORDERED	92	T800	Floating point orderability
FPREMFIRST	8F	T800	Floating point remainder first step
FPREMSTEP	90	T800	Floating pt remainder iteration step
FPREV	A4	T800	Floating point reverse
FPRTOI32	9D	T800	Convert real to int32
FPSTNLDB	84	T800	Floating store non-local real64
FPSTNLI32	9E	T800	Floating point store non-local int32
FPSTNLSN	88	T800	Floating store non-local real32
FPSUB	89	T800	Floating point subtract
FPTESTERR	9C	ALL	Test fp error false and clear
FPUABS	0B	SEQ	Floating point absolute
FPUCHKI32	0E	SEQ	Check fp in range of type int32
FPUCHKI64	0F	SEQ	Check fp in range of type int64
FPUCLRERR	9C	SEQ	Clear floating point error
FPUDIVBY2	11	SEQ	Floating point divide by 2.0
FPUEXPDEC32	09	SEQ	Floating point divide by 2^32
FPUEXPINC32	0A	SEQ	Floating point multiply by 2^32
FPUMULBY2	12	SEQ	Floating point multiply by 2.0
FPUNOROUND	0D	SEQ	Conv real64 to real32 w/o rounding
FPUR32TOR64	07	SEQ	Convert real32 to real64
FPUR64TOR32	08	SEQ	Convert real64 to real32
FPURM	05	SEQ	Set fp rounding to round to -infinity
FPURN	22	SEQ	Set fp rounding to round-to-nearest
FPURP	04	SEQ	Set fp rounding to round to +infinity
FPURZ	06	SEQ	Set fp rounding to round-to-zero
FPUSETERR	23	SEQ	Set floating point error
FPUSQRTFIRST	01	SEQ	Floating point square-root first step
FPUSQRTLAST	03	SEQ	Floating point square-root last step
FPUSQRTSTEP	02	SEQ	Floating point square-root step
GAJW	3C	16/32	General adjust workspace
GCALL	06	16/32	General call
GT	09	16/32	Greater than
IN	07	16/32	Input message
LADD	16	16/32	Long add
LB	01	16/32	Load byte
LDDEVID	17C	16/32	Load device ID
LDIFF	4F	16/32	Long difference
LDINF	71	T414	Load real32 floating point infinity
LDIV	1A	16/32	Long divide
LDMEMSTARTVAL	7E	16/32	Load "MemStart" address
LDPI	1B	16/32	Load pointer to instruction
LDPRI	1E	16/32	Load current priority
LDTIMER	22	16/32	Load timer
LEND	21	16/32	Loop end
LMUL	31	16/32	Long multiply
LSHL	36	16/32	Long shift left
LSHR	35	16/32	Long shift right
LSUB	38	16/32	Long subtract
LSUM	37	16/32	Long sum
MINT	42	16/32	Minimum integer

MOVE	4A	16/32	Move message
MOVE2DALL	5C	ALL	2D block move
MOVE2DINIT	5B	ALL	Initialize 2D block move
MOVE2DNONZERO	5D	ALL	2D block move, non-zero bytes
MOVE2DZERO	5E	ALL	2D block move, zero bytes
MUL	53	16/32	Multiply
NORM	19	16/32	Normalize
NOT	32	16/32	Boolean NOT
OR	4B	16/32	Boolean OR
OUT	0B	16/32	Output message
OUTBYTE	0E	16/32	Output byte
OUTWORD	0F	16/32	Output word
POP	79	16/32	Pop stack
POSTNORMSN	6C	T414	Post-normalize real32 fp number
PROD	08	16/32	Product
REM	1F	16/32	Remainder
RESETCH	12	16/32	Reset channel
RET	20	16/32	Return
REV	00	16/32	Reverse
ROT	79	16/32	Rotate stack
ROUNDSN	6D	T414	Round real32 floating point number
RUNP	39	16/32	Run process
SAVEH	3E	16/32	Save high priority queue registers
SAVEL	3D	16/32	Save low priority queue registers
SB	3B	16/32	Store byte
SETERR	10	16/32	Set error
SETHALTERR	58	16/32	Set halt-on-error
SETJ0BREAK	B3	16/32	Set breakpoint flag
SHL	41	16/32	Shift left
SHR	40	16/32	Shift right
START	1FF	16/32	Mostly simulate hardware reset
STARTP	0D	16/32	Start process
STHB	50	16/32	Store high priority back pointer
STHF	18	16/32	Store high priority front pointer
STLB	17	16/32	Store low priority back pointer
STLF	1C	16/32	Store low priority front pointer
STOPERR	55	16/32	Stop on error
STOPP	15	16/32	Stop process
STTIMER	54	16/32	Store timer
SUB	0C	16/32	Subtract
SUM	52	16/32	Sum
TALT	4E	16/32	Timer alt start
TALTWT	51	16/32	Timer alt wait
TESTERR	29	16/32	Test error false and clear
TESTHALTERR	59	16/32	Test halt-on-error
TESTHARDCHAN	2D	16/32	Report link engine current status
TESTJ0BREAK	B4	16/32	Test breakpoint flag
TESTPRANAL	2A	16/32	Test processor analyzing
TIMERDISABLEH	7A	16/32	Disable high priority timer
TIMERDISABLEL	7B	16/32	Disable low priority timer
TIMERENABLEH	7C	16/32	Enable high priority timer
TIMERENABLEL	7D	16/32	Enable low priority timer
TIN	2B	16/32	Timer input
UNPACKSN	63	T414	Unpack real32 floating pt number

WCNT	3F	16/32	Word count
WSUB	0A	16/32	Word subscript
WSUBDB	81	T800	Form double word subscript
XDBLE	1D	16/32	Extend to double
XOR	33	16/32	Boolean XOR
XWORD	3A	16/32	Extend to word

Appendix C: TASM Internals

Source Code Organization and Compiling

The TASM system consists of six "C" source files and four include files:

1. "tasm1.c". Top level and I/O primitives.

2. "tasm2.c". Input parsing.

3. "tasm3.c". Integer expression evaluator.

4. "tasm4.c". Prefix "binding" and optimization.

5. "tasm5.c". Output and listing generation.

6. "tasm6.c". Floating point conversion.

7. "taldef.h". Configuration include file for TASM and the other assembly language tools in the Transputer Toolset. Contains the host operating system configuration selection logic, symbolic opcode names, etc.

8. "tasmdef.h". Include file, for TASM only, which defines configuration settings.

9. "tasmtyp.h". Include file for, TASM only, which defines structure types, etc.

10. "tasmext.h". Include file for, TASM only, which defines external function and data types.

For MS-DOS source distributions the supplied "makefile" may be used with the MAKE utility to build "tasm.exe" using Microsoft "C" V6.00a or Borland C++ V2.0 (the Microsoft/Borland "C" compilers are not supplied and must be purchased separately). For Macintosh source file distributions consult the supplemental information your vendor has included with the Transputer Toolset.