TCX Transputer "C" Compiler User Guide

TCX Version 91.1
6/15/91

Contents

Introduction

Overview

TCX is a "C" cross compiler for the INMOS 16 and 32 bit Transputers. TCX doesn't include "preprocessing" facilities and is intended to run as a post pass to PP (the "C" preprocessor).

System Requirements

TCX uses lots of memory and shouldn't be used for significant jobs with less than 512K bytes of program memory available. TCX doesn't use temporary files (by default anyway), and instead manipulates program constructs up to the function level in memory.

Usage

The general form of the TCX command line is:

```
tcx <input_filename> [-options]*
```

The "input_filename" field MUST contain the name of the program being compiled (typically the output of a "C" program run through PP). Note that a default input filename extension of ".pp" is assumed unless one is explicitly provided (to match PP output files).

The "option" field allows control over the way TCX compiles programs. It contains zero or more "control flags", which collectively allow you to configure TCX to meet your requirements.

One of the allowed flags is "-v". This flag is used to toggle the compiler between the verbose and quiet output modes. Unlike other options whose actions are independent of the TCX runtime environment, the "-v" flag action is dependent on which host operating system TCX is running under. TCX on the Macintosh becomes "verbose" with "-v", while on other systems TCX becomes "quiet". For the remainder of this manual the assumption is made that the host system is NOT a Macintosh, and thus that the "-v" option causes the compiler to be "quiet". If you are using TCX on the Mac you will have to mentally reverse the commentary relating to the "-v" flag as it appears in this manual!

Examples

Assume you have a "C" program named "chello.pp" which you wish to compile (we presume it has already been processed by PP). The syntax is:

```
tcx chello
```

Since TCX is verbose by default it gives you a function by function commentary as the "chello.pp" program is compiled. To turn off the verbose output you use:

```
tcx chello -v
```

By default TCX generates code for the T414 processor type. If instead you wish to generate T800 code you use:

```
tcx chello -v -p8
```

As an alternative, you can merge the option flags together:

```
tcx chello -vp8
```

See the following **Option Description** section for information on the option flags used when compiling code for other members of the Transputer family.

In the above examples TCX wrote the output assembly language file to "chello.tal". If you wish to put it somewhere else you can use the "-o" option:

```
tcx chello -ovp8 zip.zap
```

This example causes the output to be written to a file by the name of "zip.zap". Now, assuming you are more interested in speeding things up than in debugging information, you may use the "-c" option to compress the output file:

```
tcx chello -vc
```

This example disables verbose commentary and removes the source debugging information from the "chello.tal" output file. The code is compiled for the default T414 processor.

Option Descriptions

The above examples cover some of the operational details of TCX. In addition to the following detailed option summary note that simply executing TCX without any command line arguments causes it to summarize the allowed syntax and option flags.

Note that TCX provides a return value of non-zero to the operating system if errors are detected. This may be used by some shell programs to respond to the error in a programmable fashion. If an error was detected an error message is written to standard output regardless of whether the "-v" option was specified.

The following option listing summarizes and expands upon the information presented in the examples:

`-c`

Compresses the output file by removing source debugging information. Unless the debugging information is required this option will substantially reduce the size of the output file and will also improve code optimization since source statement boundaries are no longer important to preserve in the generated code.

`-d`

Enable internal TCX debugging display. Causes a more detailed commentary to be written to standard output during the compilation process and causes the output file to contain a large amount of internal debugging information (parse trees, commented variable names, etc.). The output file generated in this fashion may still be assembled by TASM without difficulty but will be several times as large as the vanilla output file. This option is only really useful if you are debugging the compiler (or a new port of it), in which cases it is absolutely essential!

`-e`

Causes the output file to be written in the presence of detected errors in the input. This option is provided for debugging purposes only and may cause TCX to "coredump", or crash in a random fashion for some classes of input errors. Normally the output file is deleted if errors are present.

`-f0`

Causes TCX to make the length of "float" be 32 bits and "double" be 64 bits. Intermediate results in floating point expressions are promoted to 64 bits in conformance with "classic" (pre-ANSI), "C" practice.

`-f1`

Causes TCX to make the length of "float" be 32 bits and "double" be 64 bits. Intermediate results in floating point expressions are promoted to whatever the greatest size of the operands involved is. This option conforms to the proposed ANSI standard and adds additional flexibility in optimization when the programmer knows in advance which parts of a program require 64 bit math and which don't (and thus may execute faster). A useful facility in ANSI "C" when using this option is the "f" floating point constant suffix which informs "C" that the constant is of type "float" instead of the default "double". This option is enabled by default if none of the "-f?" options are given.

`-f2`

Causes TCX to make the length of both "float" and "double" be 32 bits. Intermediate results in floating point expressions are always 32 bits long. This option is included for those who need floating point but for whom speed is more important than accuracy. Some robotics and graphics applications seem to naturally fit this mold. When using this option you must arrange to have the preprocessor (PP), macro symbol "DOUBLE32" defined prior to any use of the various "C" include files in your program. This ensures the 32 bit versions of the various floating point constant definitions in the include files are used instead of the default 64 bit flavors. One easy way to do this is to use a "-dDOUBLE32" command line argument to PP.

`-fr`

Causes TCX to generate code which forces floating point to integral conversions to be done with "round-to-nearest" behavior. This contrasts with the standard ANSI "round-to-zero" approach. Slightly faster code is generated when this option is selected for the Transputer, and it can also be useful when doing certain types of numerical programming.

`-i`

Causes TCX to use a temp file to cache compiler intermediate information in order to reduce TCX memory requirements and allow compilation of larger "C" functions. This option has the down side of significantly increasing compiler execution time. The temp file used by the cache is located in either the current directory or in a special temporary directory associated with the "TMP" environment variable (if it exists).

`-mc<number>`

Sets the default module "number" to write the code into. The module number may be between 0 and 255 inclusive with 0 being the default. See the TASM documentation for a more detailed description of what a "module" is (really nothing more than a controllable load region).

`-mi<number>`

Sets the default module "number" to write initialized data into. The module number may be between 0 and 255 inclusive with 0 being the default. See the TASM documentation for a more detailed description of what a "module" is (really nothing more than a controllable load region).

`-mu<number>`

Sets the default module "number" to write uninitialized data into. The module number may be between 0 and 255 inclusive with 0 being the default. See the TASM documentation for a more detailed description of what a "module" is (really nothing more than a controllable load region).

`-o <output_filename>`

This option flag allows you to explicitly specify the output filename (including extension), for TCX to use for the assembly language output file. By default the output file is written on a filename constructed from the prefix of the input file with an extension of ".tal".

`-p0`

This option flag tells TCX to generate code which ONLY contains instructions common to all the various 32 bit Transputers. The intended use is the compilation of machine independent code. In actual fact what this does is shove the job off to TASM which generates subroutine calls for instructions which aren't supported in all the 32 bit processor types. To use this option a special library may be required (depending on what instructions your program requires). This library is not currently provided with the **Transputer Toolset**.

`-p2`

This option flag tells TCX to generate code for the T212/T222 processors. Note that this version of the **Transputer Toolset** does not support floating point for the 16 bit Transputers such as the T212/T222/T225. If any of the standard include files are to be used you should ensure that the preprocessor symbol "T212" is defined so that the correctly sized versions of the various include file constants are used. One easy way to do this is to use a "-dT212" command line argument to PP.

`-p25`

This option flag tells TCX to generate code for the T225 processor. Note that this version of the **Transputer Toolset** does not support floating point for the 16 bit Transputers such as the T212/T222/T225. If any of the standard include files are to be used you should ensure that the preprocessor symbol "T212" is defined so that the correctly sized versions of the various include file constants are used. One easy way to do this is to use a "-dT212" command line argument to PP.

`-p4`

This option flag tells TCX to generate code for the T414 processor. This is also the default if none of "-p?" flags is given. If any of the standard include files are to be used you should ensure that the preprocessor symbol "T414" is defined so that the correctly sized versions of the various include file constants are used. One easy way to do this is to use a "-dT414" command line argument to PP.

`-p45`

This option flag tells TCX to generate code for the T400/T425 processors. If any of the standard include files are to be used you should ensure that the preprocessor symbol "T414" is defined so that the correctly sized versions of the various include file constants are used. One easy way to do this is to use a "-dT414" command line argument to PP.

`-p8`

This option flag tells TCX to generate code for the T800 processor. If any of the standard include files are to be used you should ensure that the preprocessor symbol "T800" is defined so that the correctly sized versions of the various include file constants are used. One easy way to do this is to use a "-dT800" command line argument to PP.

`-p85`

This option flag tells TCX to generate code for the T801/T805 processors. If any of the standard include files are to be used you should ensure that the preprocessor symbol "T800" is defined so that the correctly sized versions of the various include file constants are used. One easy way to do this is to use a "-dT800" command line argument to PP.

`-pc`

Make a plain "char" be signed (defaults to unsigned). Either way is legal under ANSI but unsigned generates somewhat faster code.

`-ps`

Make signed right shifts propagate the sign. TCX zero fills by default (which is faster). Either way is legal under ANSI.

`-q0`

Tells TCX to disable the internal post-pass code optimizer. By default TCX performs this additional optimization.

`-q1`

Tells TCX to disable the inline code expansion for "intrinsic" functions. See the **"C" Intrinsic Functions and Language Extensions** section for more information.

`-q2`

Tells TCX to disable the the CPU/FPU instruction concurrency optimization normally performed when compiling code for Transputers with floating point support.

`-r`

Tells TCX to generate runtime relocatable code. This forces all references to statically allocated structures to be PC relative. As usual, you must ensure that you don't have pointers with static initializers, explicit casts to literal addresses, etc. Additionally, you must ensure that all other files which make up the final program ALSO use this option to ensure correct operation after relocation. This includes all required library routines (the distribution versions of "t2lib.tll", "t4lib.tll"/"t432lib.tll" and "t8lib.tll"/"t832lib.tll" are NOT relocatable, you must rebuild a version for this purpose). A slight drawback to this option is that the generated code is usually somewhat larger.

`-s`

This option is similar to the "-r" option except it allows separate relocation of code and data at runtime. This option implements what INMOS terms the "static link" data model. This means that the parameter list to each function which is called contains a hidden "link" to the current global memory base address. All accesses to global data within a function are then relative to this base address. This option is designed to work in conjunction with the "S" TLNK option flag. TLNK provides the global definition of the static base symbol and delays the binding of the T_ADDR_DATA so that runtime fixups can be implemented when the resulting ".tld" file is loaded/located.

This option also modifies the settings of the following other TCX options:

1. Enables the "-r" flag so that code is relocatable.
2. Forces code into load module 0 (equivalent to "mc0").
3. Forces initialized data into load module 250 (equivalent to "mi250"). Note that TLNK will later predefine the static link base symbol ("?slb"), at the beginning of this load module.
4. Forces uninitialized data into load module 252 (equivalent to "mu252").

The libraries shipped with the **Transputer Toolset** will NOT work with the static link data model without recompilation. Although the library source code has been modified to correctly work with this model (note that a preprocessor symbol called "STATIC_LINK" must be defined when compiling the library code for this mode), the library routines which initialize new processes must be modified to initialize the static link thread. This includes routines such as "_main", "ProcPar", etc. As the primary use of the static link option is with the various Transputer operating systems which require it (and have diverse ideas of how process creation should be handled), this is generally handled by OS specific library routines.

`-ti<function_name>`

This option flag tells TCX to generate a "call", at the beginning of each function, to the user specified "function_name" function. Use of this option permits the creation of various forms of execution "tracing", including stack tracing. The specified function to be called is passed a single integer parameter which contains the workspace size (in words), for the function it was called from. By adding the workspace size to the workspace pointer of the calling function, the address of the return address of the calling function may be determined. See the **Calling Conventions** section for more information about stack frame organization. Also see the "-to" option flag for a related facility. Note that you must NOT use the "-ti"/"-to" flags when you are compiling the trace functions proper (otherwise they will recurse to death tracing themselves!) See the "trace.c" file in example program directory for an example of a function suitable for use with this option.

`-to<function_name>`

This option flag tells TCX to generate a "call", at the end of each function, to the user specified "function_name" function. The "call" is also generated just prior to any "return" statements the function may contain. Use of this option permits the creation of various forms of execution "tracing". See the **Calling Conventions** section for more information about stack frame organization. Also see the "-ti" option flag for a related facility. Note that you must NOT use the "-ti"/"-to" flags when you are compiling the trace functions proper (otherwise they will recurse to death tracing themselves!) See the "trace.c" file in example program directory for an example of a function suitable for use with this option.

`-v`

This option flag disables the "verbose" output mode which is the default for TCX. If you use this option you won't get any garbage written to standard output unless you actually have an error.

`-w0`

The "w" option flags allows you to set the warning level for TCX. The "w0" option suppresses ALL warning messages. Only error messages which result from explicit syntax or semantic errors are flagged. This level is not recommended for normal operation!

`-w1`

This version of the "w" option flag selects an intermediate level of TCX nit-picking. This level is approximately equivalent to a fairly strict UN*X-style "C" compiler. The "w1" warning level is a reasonable choice for porting existing "C" code which doesn't maintain exact type equivalance for function arguments, etc.

`-w2`

This "w" option version enables the "picky" level of TCX warning messages. Since TCX is evolving towards the emerging ANSI "C" standard many of these messages are oriented around the requirements of that flavor of "C". The "w2" warning level is the default for TCX and should be used for all new code to ensure maximum future portability to other "C" compilers (both ANSI and traditional).

Language Description

ANSI/V7 Conformance

There are several possible models of what a "C" compiler should be. PP (now) and TCX (eventually) are intended to conform to the Proposed ANSI "C" Standard. Since PP is documented elsewhere, this section describes where TCX differs from the ANSI standard. Note that discounting the vagueness of the original specification, TCX is completely upward compatible with UN*X V7.

ANSI things TCX doesn't CURRENTLY support:

1. The internationalized character extensions.
2. "const" and "volatile".

Major ANSI things TCX does CURRENTLY support:

3. Signed and unsigned bitfields.
4. Complex auto initializers.
5. Structure passing and return.
6. Both 32 and 64 bit IEEE floating point.
7. Function prototypes.
8. Separate name spaces for structure components.
9. Structure assignment.
10. String concatenation.
11. Enumeration specifiers.
12. "void" and "void *".
13. Union initializers.

For a good description of the "C" language (and the Proposed ANSI "C" Standard), the following references are recommended:

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

The "C" Programming Language (Second Edition)
Brian W. Kernighan/Dennis M. Ritchie
Prentice-Hall, Inc.
Englewood Cliffs, NJ 07632

Library Notes

In practice the "C" compiler is only half the battle, a good library is nearly as important! At its current stage of development TCX doesn't have everything which will eventually be present, but it does offer a rich set of primitives. All the routines in the library which are part of the ANSI specification conform to ANSI dictates. Those routines which aren't part of ANSI but are common in "C" libraries conform to the SYS5 specification or the BDS4.3 specification (whichever applies). See the **TRANSPUTER 'C' LIBRARY DESCRIPTION** manual for detailed information.

Five precompiled versions of the "C" library are provided with this **Transputer Toolset** release:

```
"t2lib.tll"    For the T212/T222/T225 processors.

"t4lib.tll"    For the T400/T414/T425 with 64 bit "double".
"t432lib.tll"  For the T400/T414/T425 with 32 bit "double".

"t8lib.tll"    For the T800/T801/T805 with 64 bit "double".
"t832lib.tll"  For the T800/T801/T805 with 32 bit "double".
```

Note that libraries where both "float" and "double" are 32 bits have been compiled with the "-f2" option flag. These libraries should only be used with user programs compiled in the same manner. Only one set of include files is provided, but the size of the "double" type may be controlled using the "DOUBLE32" macro. If the size of a "double" is to be 32 bits you should ensure that a macro named "DOUBLE32" is defined BEFORE the first include file is used in your program. This allows the include files to determine which of the 32 or 64 bit versions of the various floating point constants they contain should be used. Set the "-f2" commentary in the **Option Descriptions** section for more information.

Data Representation and Alignment

For the 16 bit Transputers, the following are the sizes of the basic data types:

pointer	16 bits (signed)
bitfield	16 bits (field sizes from 1 to 16 bits)
enum	16 bits
char	8 bits
short	16 bits
int	16 bits
long	16 bits (may be 32 bits in a future version)

For the 32 bit Transputers, the following are the sizes of the basic data types:

pointer	32 bits (signed)
bitfield	32 bits (field sizes from 1 to 32 bits)
enum	32 bits
char	8 bits
short	32 bits
int	32 bits
long	32 bits
float	32 bits
double	32 or 64 bits depending on "-f?" option flag
long double	32 or 64 bits depending on "-f?" option flag

By default, "char" is unsigned (the "-pc" option flag may be used to make "char" signed). By default, right shifts of signed integers zero fill (the "-ps" option may be used to force sign propagation).

All "static/global" data objects other than "char" (or "array of char"), are word aligned. All "auto" data objects are word aligned. The size of all structure and union data objects is padded to the nearest word boundary.

"C" Intrinsic Functions and Language Extensions

The Transputer has a relatively rich instruction set with facilities for math and process scheduling built into the microcode. One of the standard problems with taking advantage of this sort of thing, in an existing language like "C", is finding a good way to use these facilities WITHOUT compromising future application program portability. Two basic approaches have been taken for the Transputer:

1. Extend the syntax of "C" to support the facilities available on the Transputer.

2. Build a set of library routines to access the features and thereby keep the language definition standard.

Approach #1 provides a fairly clean interface to the Transputer extensions such as channel I/O, scheduling, etc., but severely hampers future portability. Approach #2 provides a somewhat "clunky looking" interface to the Transputer features and has the added burden of a function call overhead tacked onto the intrinsically low overhead Transputer instructions.

This is the quandary we faced during the design phases for the TCX compiler: Which way should WE go?

In the end, the portability argument won out. One particularly strong point was made when someone pointed out that we had all, at one time or another, done maintenance on programs which were almost as old as we were (written long before the Transputer was a gleam in anyone's eye).

Given this decision, we decided to "cheat" a bit by implementing the desired functions as both real functions and as "intrinsics", which are specially recognized by TCX, and for which inline code is generated. This balanced the efficiency scales and left us with a tradeoff between portability and code appearance (which really isn't that bad anyway). In light of this, the following functions were implemented as "intrinsics" (see the **TRANSPUTER 'C' LIBRARY DESCRIPTION** manual for details):

T225/T400/T425/T800/T801/T805 only:

```
BitCnt    BitRevNBits         BitRevWord
```

T400/T425/T800/T801/T805 only:

```
Move2D    Move2DNonZero       Move2DZero
```

T800/T801/T805 only:

```
fabs      fabsf               sqrt          sqrtf
```

All processors:

```
bcopy     ChanIn              ChanInChar    ChanInInt
ChanOut   ChanOutChar         ChanOutInt    ChanReset
GetLoPriQ GetHiProQ           memcpy        PHalt
ProcAfter ProcGetPriority     ProcReschedule ProcStop
ProcWait  PRun                PStop         SetLoPriQ
SetHiPriQ SetTime             Time
```

To allow a function to be recognized as an "intrinsic", the include file which contains the function prototype definition for the desired function must be used (so TCX knows about the function). If you wish to ensure that you get a real function call, and not the inline equivalent, you may simply "#undef" the name of the function and the "intrinsic" recognition will be replaced with a real function call (ala ANSI).

Calling Conventions

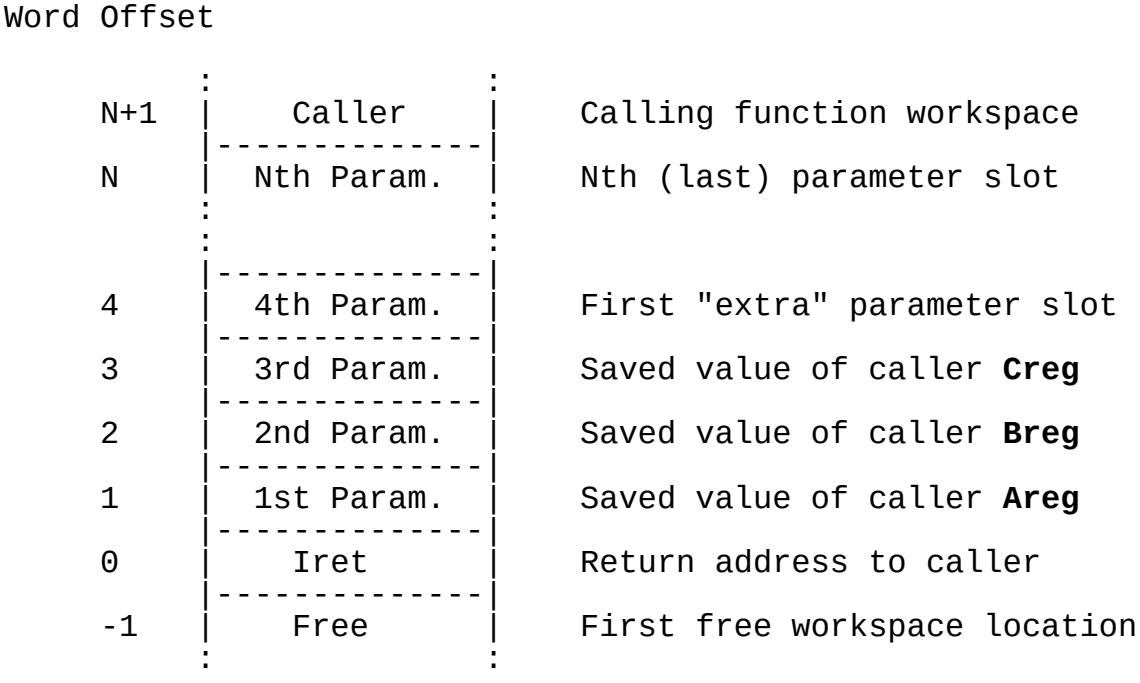
This section describes the entry and exit conditions for a "C" function compiled by TCX. Consult the appropriate INMOS documentation for further information on the instruction-level architecture of the Transputer.

Call State

1. All floating point and integer registers are undefined.

2. **Iptr** points to the first instruction in the function being called.

3. **Wptr** points to the base of the invocation stack frame prepared by the calling function. **Wptr** must always be word aligned. The invocation stack frame has the following format (addresses decrease from top to bottom):



Note that at least 3 parameter slots (words), are used, in addition to the return address, regardless of the size or number of actual parameters. These correspond to the locations into which **Areg**, **Breg** and **Creg** are automatically stored when the "call" instruction is issued.

The use of the parameter slots depends on whether the function will be returning a structure or union as well as the global data model being used by TCX. If the function is returning a structure or union the "1st Parameter" slot will ALWAYS hold the address to copy the return value into. If the "static link" global data model is in use the static link will be contained in either the "1st Parameter" or "2nd Parameter" (if the function returns a structure or union), slot.

If either the "1st Parameter", or both the "1st Parameter" and "2nd Parameter", slots are already in use, any parameters which would otherwise use these slots will be bumped up by the corresponding number of slots.

If one or more of the default three parameter slots are not used by a particular function call then the slot is available for use by the called function to store local variables, etc.

Parameters

All parameters passed by the calling function start at the first available parameter slot and continue upwards (towards higher addresses). Parameters are placed in the slots in the order in which they are present in the original "C" source description. The first location in the local workspace used by the calling function is located in the next slot above the last parameter passed to the called function.

All passed parameters occupy at least one slot. All parameters are passed by value. Parameters which are larger than one word occupy the minimum number of slots (words), required to hold the respective data objects. The "endian-ness" of the memory layout (for data items which require more than one slot), is the same as that normally used for local or global variables. See the **Data Representation and Alignment** discussion for information about the sizes of objects.

Return Values

For functions which return structures or unions, the passed return address is used as the address to copy the return value into. It is the responsibility of the called function to perform the copy.

For functions which return integral types, the return value is passed in **Areg**. On Transputers with floating point support, floating point return values are passed in **FAreg**.

Floating point return values, on Transputers without floating support, (no hardware **FAreg**), are copied into a simulated **FAreg** register. This simulated register is the "FAreg" member, of the "FPstate" structure, which is used by the floating point emulation library to simulate a set of floating point registers. It is the responsibility of the called function to copy the return value into this structure member. An external definition for the structure is contained in the "conc.h" include file.

Return State

The called function is responsible for restoring the workspace pointer to the value it was on entry to the function, and executing a "ret" instruction to return control to the calling function. The calling function has the responsibility of adjusting the workspace to remove any parameters beyond those automatically removed by the "ret" instruction.

Inline Assembly Language

TCX supports inline assembly language. For a description of the Transputer assembly language see the TASM assembler documentation. For a description of the Transputer instruction set and architecture, consult the various INMOS publications.

WARNING: Programming the Transputer at the assembly language level is not for the faint at heart! You must have an excellent understanding of the low-level hardware and software features of the Transputer to have a reasonable chance of success!

From the point of view of TCX, inline assembly code is passed straight through to the assembler without modification or examination. Keep this in mind and ensure that what you do will not disrupt what TCX is doing (or vice-versa). To ensure maximum compatibility between the two, follow these rules:

1. If necessary, precede your inline assembly code with a "C" null statement (";"), in order to ensure that the last thing that was parsed prior to your inline code is a simple "C" statement.

2. Make sure that the workspace pointer is unchanged between the start and end of the inline assembly code. If you must temporarily change the workspace pointer within the assembly code, use the "_asm_ajw" pseudo-function described later.

3. Never change the contents of anything in the local workspace without reason. If you do access local variables, or parameters, use the "[]" notation used in the examples below to compute the workspace offset. This is particularly important for portability, since different versions of TCX may assign different workspace offsets to the same variables (as optimization improves). Note that the workspace offsets for the structure/union return value pointer and static link pointer may also be accessed using this technique by using the reserved symbols "?rslink", and "?slb", respectively.

4. Whenever possible, use the "_asm_eval?" pseudo-functions described below to evaluate "C" expressions or addresses. Again this will greatly improve portability between different versions of TCX.

5. Whenever possible, make the inline assembly language be the only code in a function. This causes the fewest possible conflicts between TCX and what you are doing.

6. When things don't work as you expect, examine the assembly language output from TCX to see what is REALLY being generated. As mentioned earlier, inline assembly coding isn't for the casual programmer!

You invoke the inline assembly language option by issuing a "#pragma asm" statement. The option is turned off by a complementary "#pragma endasm" statement. For example:

```
int  add(int a, int b)
     {
     ;
#pragma   asm
     ldl  [a]
     ldl  [b]
     add
#pragma   endasm
     }
```

This function takes two integer arguments, adds them, and returns the result on the top of the integer stack (in Areg). See the previous **Calling Conventions** section for more information about function entry/exist conventions.

Of course, for this simple case the code is no better than the "C" compiler can generate. In general, the best use of inline assembly is to take advantage of instructions and facilities the Transputer supports which are not the normal provenance of the "C" language. Keep in mind that many of the more useful facilities already have been supported with "intrinsic" functions or library routines.

The preceding example also showed how the "[]" notation is used to allow assembly instructions symbolic access to the workspace offset for "auto" variables and parameters. For non-static global variables the actual name may be used instead. For example:

```
int  a,b;

int  gadd(void)
     {
     ;
#pragma   asm
     .ldc a
     ldnl 0
     .ldc b
     ldnl 0
     add
#pragma   endasm
     }
```

Like the previous example, this function adds two variables (named "a" and "b"), and returns the result. If "a" and "b" were defined in another file you would have to indicate that using a ".ext a,b" pseudo-opcode.

If the variable to be accessed is more complex than a simple integer variable, you might wish to use one of the "_asm_eval?" pseudo-functions. These functions are issued with "C" syntax, but may be embedded inside inline assembly language. For example:

```
#include  <inline.h>

int  a,b;

int  gadd(void)
     {
     ;
#pragma   asm
     _asm_eval2(a,b);
     add                 ;"a" in Areg, "b" in Breg
#pragma   endasm
     }
```

Note the required include file "inline.h". What the "_asm_eval2" pseudo-function does is to evaluate arbitrary "C" expressions into the Transputer **Areg** and **Breg** (integer stack), registers. The **Creg** register is left unmolested. There are also versions for evaluating only into **Areg** ("_asm_eval1"), or into all three registers ("_asm_eval3").

Although the pseudo functions may only be directly used with the integer registers, they are helpful for use with floating point instructions also. This is because most floating point instructions use the integer register to hold the address of floating point numbers being loaded and stored. For example:

```
#include  <inline.h>

float     a,b;

float fgadd(void)
     {
     ;
#pragma   asm
     _asm_eval2(&a,&b);
     fpldnlsn             ;Load "a"
     fpldnladdsn          ;Load "b" and add with "a"
#pragma   endasm
     }
```

This function simply adds the values of "a" and "b" and returns the result.

While none of these examples has required it, it is sometimes necessary to change the workspace pointer while programming in inline assembly language. The normal "ajw" instruction may be used, however, the symbolic "C" access facilities will then not work correctly. If you wish to use the "[]" notation, or the "_asm_eval?" pseudo-functions, you should replace use of "ajw" instructions with the "_asm_ajw" pseudo-function. The use of this pseudo-function keeps TCX abreast of the current workspace pointer position and allows it compute symbolic workspace offsets correctly. For example:

```
#include  <inline.h>

int  zip(int a, int b, int c)
     {
     ;
#pragma   asm
     ldl  [a]
     ldl  [b]
     add
     _asm_ajw(-1);        ;Decrement workspace pointer
     stl  0               ;New workspace temp storage
     ldl  [c]
     .ldc 1
     .ldc 2
     call @zap
     ldl  0               ;Re-load sum of "a" & "b"
     add                  ;Add to value returned by "zap"
     _asm_ajw(1);         ;Restore workspace pointer
#pragma   endasm
     }
```

Note that the "inline.h" include file is again required. In this example, "a" and "b" are added together, and the sum is temporarily saved. Then, the value of the "c" parameter and two constants are loaded onto the integer stack and the "zap" function called. Finally, the return value from "zap", and the previously saved sum, are added and returned as the value of the "zip" function.

In addition to the information presented here, there are several other sources of inline assembly language examples and ideas:

1. The "C" library. Many of the library routines use inline assembly language (for example "tcio.c" and the math functions), these functions give you many examples of how to use these features.

2. The TCX assembly language output file. Try first coding your algorithm in "C", and then using the assembly language output file as a starting point in constructing the desired inline assembly language code.

Appendix A: Error Messages

Types of Error Messages

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

- **Simple Errors**. These are used to report problems which are fatal and are not generally related to the contents of the file being compiled. This includes things like TCX not being able to open the specified input file, etc. The format is simply:

```
message_text
```

- **Compile Errors**. These are used to report problems which are usually related to the file being compiled. In addition to the error message text, TCX also displays the filename and line number where the error was discovered. The format is:

```
<filename> @ #: message_text
```

- **Compile Warnings**. These are used to report problems which are related to the file being compiled, but may or may not be fatal (depending on the setting of the "-w#" flag). As with **Compile Errors**, the filename and line number where the warning was discovered is displayed in addition to the warning message text. The format is:

```
<filename> @ #: WARNING: message_text
```

Either the **Compile Errors** or **Compile Warnings** messages may be optionally followed by a copy of the source line which produced the error with a "^" underneath showing the specific spot where the error was encountered.

Appendix B: TCX Internals

Source Code Organization and Compiling

The TCX system consists of fifteen "C" source files and four include files:

1. "tcx1.c". Block level and external parsing.
2. "tcx2.c". Function parsing.
3. "tcx3.c". Control structure parsing.
4. "tcx4.c". Expression parsing.
5. "tcx5.c". Keyword and type parsing primitives.
6. "tcx6a.c". Code generator primitives.
7. "tcx6b.c". Code generator.
8. "tcx6c.c". Code generator.
9. "tcx6d.c". Code generator.
10. "tcx6e.c". "intrinsic" code generator.
11. "tcx7.c". Post-pass optimizer.
12. "tcx8.c". I/O and misc primitives.
13. "tcx9.c". Text block manipulation primitives.
14. "tcx10.c". Command line cracking and initialization.
15. "tcx11.c". Symbol table manipulation.
16. "tcx.h". Main configuration and data structure definitions.
17. "tcxdecl.h". Global data declarations (also global definitions when included in "tcx1.c").
18. "tcxenv.h". Host operating system and target architecture configuration.
19. "tcxext.h". External function declarations.

For MS-DOS source distributions the supplied "makefile" may be used with the MAKE utility to build "tcx.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.