







































































## 7.5 Run TAL2C

Run TAL2C. If you begin with TAL code that compiles successfully, the only errors you should expect from TAL2C concern the features of TAL that are incompatible with C. These include hardware-specific function calls (such as `$rp`), some complex equivalencing and initialization, and some zero-sized arrays.

See Chapter 8, "TAL2C Limitations," for information on things that require manual attention. See Chapter 10, "Error Messages," for error messages that TAL2C may display during conversion.

TAL2C also displays warnings about potential compatibility limitations. If, for example, you use arrays with non-zero lower bounds, or complex block move expressions, TAL2C warns you of the consequences in terms of performance and information loss. See Chapter 10, "Error Messages," for warning messages that TAL2C may display during conversion.

The following command line converts `myprog`, making use of the files in the `headers` folder, with the sections 1, 3, and 10 toggled in:

```
tal2c -I headers -t 1,3,10 myprog
```

When the conversion finishes successfully, you will have a TAL main procedure that calls the first converted C procedure. This circumvents a difference between TAL and C start-up processing procedures, which is described in Chapter 8, "TAL2C Limitations." If required, you can rewrite your start-up code in C by hand at your later convenience.

## 7.6 Examine the Resulting Messages and C Code

As you examine the resulting C code, you might find that some TAL code was not converted completely.

In particular, if your TAL code contains `DEFINES` you must run TAL2C on the TAL code again to resolve the `DEFINES`. After this second pass, you may find that TAL2C could still not convert all of the `DEFINES` completely.

If so, you should inspect the C output code and manually convert the problem `DEFINES`. You can do this by inserting into the TAL code the equivalent C in special comments that start with hyphen, hyphen, tilde (`--~`). You could also make the modifications directly to the C output code, but only if you do not intend to run the conversion again.

See "Using Comments to Control Conversion," in Chapter 9, "Fine Tuning," for details.

## 7.7 Compile the C Output Code

When you compile the C output code, you can expect to get some warnings if your TAL program uses 16-bit pointer operations and converts them to or from `INTs`. See Chapter 8, "TAL2C Limitations," for details.

You may also get some warnings if your TAL program makes extensive use of subprocs that share data with their owning proc. In this case, you may find large strings of parameters being passed between C functions. If you have a good working knowledge of the original TAL program, you may be able to eliminate many of these parameters, and significantly improve the performance of the translated code.

If you compile with Microsoft C compilers you may get warnings about redefinitions of `min` and `max`. You can safely ignore these warnings.

## 7.8 Fine-Tune Your C Output Code

After you have compiled your C code, you have a functional, portable C version of your TAL program. You may still want to do some work on it to optimize its performance. See Chapter 9, "Fine Tuning," for some tips on how best to do this.

## 8 TAL2C Limitations

There are some things that TAL2C cannot handle automatically. Most relate to hardware-specific TAL features for which C has no equivalent. Others relate to structural differences between the languages.

In some cases, you may have to modify your TAL code. Every effort has been made to keep such modifications to a minimum. In most cases, the modifications are minor syntactic distinctions to TAL, but they make a big difference to the compiler. In other cases, TAL2C displays a message if it encounters TAL code that uses complex expressions that would generate ambiguous C output code.

This chapter describes TAL2C limitations and discusses how you can work around them. It covers:

- Hardware-Specific Statements
- Hardware-Specific Functions
- Block Moves
- ?search
- ?UseGlobals and ?SaveGlobals
- ?page
- struct Types Declared in Arguments
- Equivalencing
- Process Startup
- Conversion of Function Pointer Types
- Zero-Sized Arrays
- Single-Instance struct Definitions
- SQL INVOKE Statements
- FIELDALIGN Directive
- Fixed(N) Data Type
- Retyping Pointer Variables
- External Anonymous Structures
- External Declarations for Variable Procs
- pTAL Considerations

## 8.1 Hardware-Specific Statements

The following statements are hardware-specific, and, therefore, are not supported:

CODE

STACK

STORE

## 8.2 Hardware-Specific Functions

The following functions are hardware-, language-, or operating-system specific, and, therefore, are not supported:

`$overflow`

`$rp`

`$type`

`$usercode`

The `$bitlength` function is supported only when referencing an object that is a multiple of eight bits in size. TAL2C substitutes `8*sizeof()`.

The `$bitoffset` function is supported only when referencing an object on a byte boundary. TAL2C substitutes `8*offsetof()`.

The `$carry` function is supported only in conjunction with `scan` (other uses are hardware specific).

The `$point` function is supported, but because of the nature of the `FIXED` data type, some information is lost.

## 8.3 Block Moves

In block moves consisting of array constants with repetitive literal elements, the literal is lost. The C output code runs correctly, but it can be more difficult to maintain. Additionally, there could be some loss of performance.

TAL2C creates global variables-one for each basic data type-that are used in mimicking block copies of constants. You may want to consider other ways of implementing this, depending on your program's requirements.

TAL2C tries to replace block copies with `memcpy`s whenever it can, to improve performance, because many C compilers optimize `memcpy`s very well. This can lead to problems because block copies work well for overlapping memory regions; `memcpy`s, however, do not. You can work around this by replacing the `memcpy` with a call to a C library function that handles overlapping memory such as `memmove`.

## 8.4 ?search

This compiler directive allows the use of an object file as a sort of header. For TAL2C to correctly interpret and convert the rest of your code, you must include the source code that was used to generate the object.

## 8.5 ?UseGlobals and ?SaveGlobals

These TAL-specific compiler directives allow global definitions to be put into a symbol file. For TAL2C to correctly interpret and convert the code, you must include the original source code from the TAL run that was used to produce the symbol file.

## 8.6 ?page

This compiler directive is subject to the same positional constraints as comments when it is converted to `#pragma page` in C.

## 8.7 struct Types Declared in Arguments

The construction below is not C-compatible, and results in a C compiler error ("undeclared struct type").

```
proc a(s)
  struct.s
  begin
    int a;
  end;
begin
```

To avoid this problem, you must create a structure template and then declare it.

## 8.8 Equivalencing

C does not support equivalencing to hardware registers or offsets to equivalents (unless the offset is into the middle of an object), because they are specific to the Tandem architecture. They can be simulated in C, but the resulting code is not portable.

```
int a;
int a1 = a; --allowed
int b = a[2]; --not allowed
int c = a - 2; --not allowed
int d = "d" - 3; --not allowed
```

## 8.9 Process Startup

C and TAL start-up processing techniques are very different. For C, Tandem-provided code handles the startup protocol and initializes context for some C library functions, for example, memory management and file I/O. For TAL, the user must provide code to handle the startup protocol and so on.

TAL code that is converted to C must normally be reworked to accommodate this difference in startup processing. However, for your convenience, TAL2C provides a dummy main procedure that you can use to invoke your converted C. The TAL source for this procedure is in the `t2cmains` file. Using this dummy startup procedure provides a quick and convenient way to run your converted C programs on your Tandem system without having to consider differences between the TAL and C runtime environments.

You can rewrite the start-up code in C at your later convenience. This is mandatory if you ever make manual adjustments to your converted C code to take advantage, for example, of standard C memory management and file I/O library functions.

## 8.10 Conversion of Function Pointer Types

Function pointers in TAL are simply `INTs`, making them hard for TAL2C to identify, while pointers in the Tandem C large or wide memory model are their own type and are always 32-bit. Mistyped pointers cause a compilation warning. One solution is to manually adjust the function pointer type in the C output code.

Another way to address this issue is to use the `function_ptrs` configuration option in the `.ctrans` configuration file. This requires you to reserve identifiers for variables in the TAL code that hold function pointers, and to be very careful not to reuse them for anything else.

## 8.11 Zero-Sized Arrays

TAL2C converts zero-size arrays to arrays of a single element. This changes the binary layout of the structure, which can cause interoperability problems unless it is the last element in the structure. There is no single way around this limitation; the solution varies with your application. Here is an example of the offending code:

```
int a[0:-1];
```

## 8.12 Single-Instance struct Definitions

Single-instance `struct` definitions are not automatically moved out of the proc to be shared by subprocs. In the following code sample, the C output code will not compile.

-- Original TAL --	/* C output code */
<pre>proc a; begin struct xyz;   begin     int member;   end;  subproc b;  begin  end; -- start proc a main code call b; end;</pre>	<pre>void a() {   struct   {     short member;   } xyz;   /* start proc a main code */   b(&amp;xyz); }  static void b(struct xyz *xyz) /* problem: xyz not global def */ {   int i;int i;   i := xyz.member;i = xyz-&gt;member; }</pre>

A solution is to change the TAL code to identify and extract the definition and mark it as a TAL structure template before the actual conversion, as the following code sample shows:

-- Fixed TAL --	/* C output code */
<pre> struct xyz_t (*); begin   int member; end;  proc a; begin   struct xyz(xyz_t);    b(&amp;xyz);  subproc b;  begin   int I;   i := xyz.member; end; -- start proc a main code call b; end; </pre>	<pre> struct xyz_t {   short member; };  void a() {   struct xyz_t xyz;   /*start proc a main code */ }  static void b(struct xyz_t *xyz) /*ok now*/ {   int i;   i = xyz-&gt;member; } </pre>

**Note:** The TAL2C places subprocs ahead of procs in order to facilitate C typing; however, they are in the other order here for clarity.

### 8.13 SQL INVOKE Statements

SQL `INVOKE` statements do not work the way you might expect, because TAL2C does not have access to the SQL catalog. To correct this, you must use `INVOKE` from `SQLCI` and then `?source` the TAL structure into the program. After the conversion, you can remove the C `#include` statement.

TAL2C recognizes the SQL extension of `/keyword UNSIGNED/` in variable declarations, and the converted C variable is declared as unsigned.

### 8.14 FIELDALIGN Directive

This compiler directive involves byte-padding; Tandem C requires even-byte boundaries for structures within other structures, standard C does not. TAL2C supports the ANSI/ISO standard whenever it can. TAL2C recognizes and ignores this directive.

### 8.15 Fixed(N) Data Type

This is a 64-bit integer with an implied decimal point. If your code uses the `FIXED` data type, you must use a C compiler that supports 64-bit integers.

### 8.16 Retyping Pointer Variables

TAL's weak typing (compared to C) and interchangeability between `INTs` and pointers can lead to extensive casting in the C output code. For example, pointers declared as `int(32)s` end up as `long`, not pointers. You can work around this by using the `function_ptrs` configuration option in the `.ctrans` configuration file.

Additionally, TAL programs can use both 16-bit (.) and 32-bit (.ext) pointers. TAL2C changes all pointers in the C conversion into 32-bit pointers in either the large or wide memory model.

TAL does not have strong type checking, so pointers to different types of data can be assigned without any explicit conversion. C differentiates between pointers based on the type of data being pointed to and requires an explicit coercion to convert between types. TAL2C generates the appropriate "typecast" code to convert between these types, but it can look unwieldy. Although runtime performance is not affected you may want to look at these sections of code to improve their appearance.

TAL often forces programmers to represent pointers as `INT(32)s`. One of the benefits of converting to C is the much stronger pointer typing.

This can eliminate many bugs at compile time rather than at run-time. Therefore, it is advisable to manually convert all pointers that were stored as `INT(32)s` in the TAL source code into appropriately typed C pointers in the C code. You should do this with considerable care if the variable is used in arithmetic expressions, because adding a variable to a pointer in C actually adds the following:

```
value * sizeof(*pointer /* what is pointed to */)
```

rather than value in TAL.

For details on pointers, see the discussion of the `ptrs` configuration section in Chapter 4, "Configuring TAL2C."

## 8.17 External Anonymous Structures

TAL2C duplicates external anonymous structures rather than introducing a new type. If you do not wish to maintain this duplication in the C output code, you must create the new type in the TAL code.

## 8.18 External Declarations for Variable Procs

If a variable or extensible proc has an external or forward declaration, the parameter names must match those in the proc definition.

## 8.19 pTAL Considerations

TAL2C supports some pTAL-related keywords added to later versions of TAL.

The pTAL data types `BADDR`, `WADDR`, `EXTADDR`, `SGBADDR`, `SGWADDR`, `SGXBADDR`, `SGXWADDR`, `CBADDR` and `CWADDR` are supported in a similar way to the way they are supported by some versions of TAL. `EXTADDR` is treated as `INT(32)`, the others are treated as `INT`; and variables with these types are treated as though they contain pointers, as if they had been specified in the `ptrs` section of the `.ctrans` configuration file.

Additionally, TAL2C supports the following pTAL type conversion functions:

```
$extaddr  
$baddr_to_extaddr  
$extaddr_to_baddr  
$baddr_to_waddr  
$waddr_to_extaddr  
$extaddr_to_waddr  
$waddr_to_baddr
```

Finally, TAL2C recognizes and ignores the `?DO_PTAL_ON` and `?DO_PTAL_OFF` directives.

## 9 Fine Tuning

The performance of the C output code can vary tremendously, depending on the style of coding and the features of the TAL language that you have used. You can dramatically improve performance by changing the code in the ways suggested in this chapter. For C that is destined to run on a Tandem computer system, you can gain even better performance by using the native Tandem RISC compiler. This chapter discusses how to fine tune the C output code produced to improve performance. It covers:

- Parameter Passing
- Tandem C Extensions
- Unsigned Bit Arrays
- Operations on Unsigned longs
- Non-Zero Lower Bounds
- Unions
- Block Moves and memcpys
- Overriding the Automatic Define Conversion
- Improving the Layout of the C Code
- Using Comments to Control Conversion

### 9.1 Parameter Passing

A major difference between TAL and C is that TAL supports nested procedure definitions (subprocs), while C does not. TAL2C converts the nested subproc definitions into separate global level procedure definitions outside the scope of the parent procedure. This is a complex process because TAL subprocs can access and modify all variables local to the procedure.

To achieve the same end in C, TAL2C explicitly passes local variables to the new global level procedure. TAL2C calculates exactly the set of variables that must be passed to each subproc either for its use, or for the use of any other subproc called within it. It also passes the parameters either by reference or by value, depending on whether the variable is modified within the subproc.

By default TAL2C passes each variable as a separate parameter to the subproc(s). This process can lead to large parameter lists if the TAL code is written to depend heavily on global access between subprocs within a procedure. If necessary, you may want to clean this up manually by looking at the way local variables are used within subprocs, and removing the unnecessary ones. Alternatively you can use the `create_struct_for_subproc` flag to have the necessary variables passed in a single structure.

### 9.2 Tandem C Extensions

Tandem provides many non-standard extensions to its C compiler, most of which are to interface to the operating system procedures written in TAL (or pTAL). When possible, TAL2C tries to avoid using these extensions to increase the portability of the resulting code. If you are running in a Tandem environment and portability is not a concern, you can keep some of the TAL-specific functionality by making use of these extensions.

### 9.3 Unsigned Bit Arrays

If the TAL code uses arrays of `unsigneds`, then the C code produced may be less than optimal. You may want to review the converted output and adjust the original TAL.

### 9.4 Operations on Unsigned longs

With a non-native Tandem C compiler, many operations on unsigned `longs` result in a library function call. This can negatively affect the performance of the C code. This problem may or may not persist with a native Tandem C compiler.

### 9.5 Non-Zero Lower Bounds

If the code uses arrays with lower bounds that are non-zero, there may be a performance penalty of a subtraction operation on every array access. If you use a lower bound of zero, then significant performance gains can be made.

### 9.6 Unions

If you plan to run the converted code on a Tandem system you may want to check whether the Tandem compiler has changed unions declared within functions into indirectly accessed data. This can slow down the C code. If these unions are the result of TAL equivalencing, they can be removed by use of the cast operator in C.

### 9.7 Block Moves and `memcpy`s

TAL block move operations are turned into `memcpy`s. This involves a function call overhead which can slow down the C output code for small `memcpy`s. To speed this up you can use `#pragma inline` and `#pragma optimize 2` in the C output code. This causes the C compiler to expand the `memcpy` function inline and to optimize the code.

There is still a performance penalty for this because the code generated by a C compiler is not as good as a single block move instruction. It is only worth optimizing converted block moves using small buffers. This is because the overhead of the procedure call becomes less significant as the size of the buffer increases.

### 9.8 Overriding the Automatic Define Conversion

Sometimes TAL2C cannot automatically convert a `define`. To explicitly convert such a `define` you may need to embed a comment starting with "`~`" in the TAL code directly after the point of definition for the `define`. For example:

```
define trace^resize^data =
string .tr^rec[0:(tr^record^hdr^len +
  tr^resize^record^len - 1)];
string .ext tr^record^hdr(tr^record^hdr^def);
string .ext tr^resize^rec(tr^resize^record^def)
#;
--~ char tr_rec[TR_RECORD_HDR_LEN + TR_RESIZE_RECORD_LEN];\
--~ struct tr_record_hdr_def *tr_record_hdr;\
--~ struct tr_resize_record_def *tr_resize_rec
```

You can also use this to handle data types where the default conversion from TAL to C is not the desired conversion. For example:

```
int function^ptr;
...
```

```
function^ptr := @proc^name;
```

would normally be converted into

```
short function_ptr;  
...  
function_ptr = proc_name;
```

which would result in an error because the types do not match.

To work around this you could do the following in the TAL code:

```
define F_PTR = int function^ptr#;  
--~ void (*function_ptr) ()  
F_PTR;  
...  
function^ptr := @proc^name;
```

This would result in the correct type being placed in the C code. After the conversion has completed, you can remove these constructs.

Additionally, specifying `--~@` always expands the define inline and does not try to turn it into a define call. For example, the following code segment

```
define fred(i) = i + count#;  
...  
i := fred(i);  
normally produces  
#define FRED(i) i + count  
...  
j = FRED(j);
```

However, the following code segment

```
define fred(i) = i + count#;  
--~@  
...  
j := fred(j);
```

produces

```
#define FRED(i) i + count  
...  
j = j + count;
```

## 9.9 Improving the Layout of the C Code

After TAL2C finishes you may want to change the layout of the C code. Gresham provides a "pretty print" program called `indent` for formatting C code. It lets you format your C code (indentation, spacing, and so on) according to your organization's coding standards.

`indent` is distributed under the terms of the GNU General Public License as published by the Free Software Foundation. You can download both the source and executable code from the Gresham web site.

## 9.10 Using Comments to Control Conversion

TAL2C allows you to insert special comments in the TAL source code to control some aspects of the conversion process. Using comments to control conversion is helpful if you intend to continue to update the TAL source code and periodically convert the TAL code to C.

By default, special comments start with hyphen, hyphen, tilde (`--~`). The location at which special comments are placed is subject to the same constraints as ordinary comments. Comments that begin with an exclamation mark (!) are always treated as plain comments.

TAL comments in the form hyphen, hyphen, tilde (`--~`) explicitly convert a define that TAL2C is unable to automatically convert. See "Overriding the Automatic Define Conversion" for details.

TAL comments in the form hyphen, hyphen, tilde, at sign (`--~@`) always expand a define inline, without trying to turn it into a define call. See "Overriding the Automatic Define Conversion" for details.

TAL comments in the form hyphen, hyphen, tilde, dollar sign (`--~$`) are inserted as is (after the `--~$` are stripped) into the C code.

TAL comments in the form hyphen, hyphen, tilde, number sign (`--~#`) are treated as toggles preventing the output of the surrounded code from being output to the C code. These are called delete toggles.

TAL comments in the form hyphen, hyphen, tilde, percent sign (`--~%`) toggle the treatment of casting enums to short \* (see option `--enum_nocast`).

TAL comments in the form hyphen, hyphen, tilde, "create\_struct\_for\_subproc" determine the use of structures to pass variables to subprocs (see option `--create_struct_for_subproc`).

TAL comments in the form hyphen, hyphen, tilde, "use\_local\_struct\_name" set the name of the structure used in passing variables to subprocs (see option `--use_local_struct_name`).

TAL comments in the form hyphen, hyphen, tilde, "expand\_array\_base\_on" generate the index of an array with a non-zero lower bound as the actual value rather than `LOWER_BOUND_DEFINE_NAME - 1`.

TAL comments in the form hyphen, hyphen, tilde, "expand\_array\_base\_off" generate the index of an array with a non-zero lower bound as `LOWER_BOUND_DEFINE_NAME - 1` rather than the actual value.

TAL comments in the form hyphen, hyphen, tilde, "begin\_create\_version\_procs" treat the following TAL source as version procedures (vprocs) and allows the `~` as a legal identifier character.

TAL comments in the form hyphen, hyphen, tilde, "end\_create\_version\_procs" treat the following TAL source normally.

TAL comments in the form hyphen, hyphen, tilde, "begin\_not\_enums" treat following literals as separate entities and does not attempt to turn them into a set of enums.

TAL comments in the form hyphen, hyphen, tilde, "end\_not\_enums" attempts to turn a set of literals into a set of enums.

TAL comments in the form `hyphen, hyphen, tilde, "begin_case_sensitive"` uses the exact case of TAL identifiers, on their declaration, in the C output. As TAL is not case sensitive identifiers are normally all output in lower case apart from define names which are upper case.

TAL comments in the form `hyphen, hyphen, tilde, "end_case_sensitive"` converts identifiers to lower case apart from define names which are upper case.

You can use the `--~` command-line option to specify a character other than the tilde (see Chapter 5, "Executing TAL2C," for details).

## 10 Error Messages

This chapter describes the error messages that TAL2C produces during the conversion process. It covers:

- Types of Error Messages
- Limitation Errors
- Limitation Warnings
- Significant Information Messages

### 10.1 Types of Error Messages

There are two types of error and warning messages that TAL2C produces:

- The first type, which receive detailed treatment in this manual, result from incompatibilities between TAL and C and the occasional resulting ambiguities that TAL2C detects during conversion.
- The second type mirrors error messages from the TAL compiler. Remember, if a program will not compile under TAL, TAL2C will not be able to convert it. Such messages are not listed here; consult your TAL compiler manual.

The words Identifier, Expression, in Point of Definition, Value, Operator, and Type refer to variables within the error messages. They will not appear in your error log as such. Instead the offending identifier, expression, etc. will be shown. For example, on your screen the first error below might read:

Number of parameters will cause AXCEL to choke in proc Find\_Name.

Accordingly, the error messages are listed here in alphabetical order, with the errors that begin with an Identifier in with the "I"'s and so forth.

Some significant information messages are also described.

## 10.2 Limitation Errors

**Message** "Number of parameters will cause AXCEL to choke in proc Identifier."

**Explanation** The Tandem C CISC-to-RISC accelerator crashes when a procedure has more than fifty-five parameters.

**Fix** Remove the excess parameters, restructure the TAL code, or do not use RISC.

**Message** "Cannot generate equivalence to another scope with an initializer."

**Explanation** Because of the way TAL2C handles equivalencing, you cannot make an initial assignment at the point of declaration

**Fix** Move the assignment to the first statement in the proc.

**Message** "Don't know how to generate an equivalence to the middle of an object in C."

**Fix** Rewrite the TAL code.

**Message** "Referring to registers isn't portable, you'll have to fix it by hand."

**Explanation** The converted code will run, but will not actually work.

**Fix** Find a different means of obtaining the needed value.

**Message** "Cannot generate \$len for struct containing zero sized array when dotted expression is Expression

**Fix** Change the structure variable name into a structure template name in the TAL (no effect on the TAL)

**Message** "Standard function \$overflow cannot be converted, re-code."  
 "Standard function \$rp cannot be converted, replacing with constant 0."  
 "Standard function \$type cannot be converted, replacing with constant 0."  
 "Standard function \$usercode cannot be converted, replacing with constant 0."

**Fix** These functions are TAL- and Tandem-specific and unavailable in C. If the constants supplied by TAL2C are insufficient, you will have to re-work your code, either in TAL or in C.

### 10.3 Limitation Warnings

<b>Message</b>	"Warning: non 0 lower bound for array Identifier, this will lead to efficiency loss."
----------------	---

**Explanation** Unlike TAL, C does not allow non-zero lower bounds. TAL2C provides code that performs the necessary subtraction, but this can lead to a loss in performance if done frequently.

**Fix** Change the code (in either language) to use a lower bound of 0.

<b>Message</b>	"Warning: arrays of unsigneds are not efficient."
----------------	---

**Explanation** TAL2C converts arrays of bit-values via DEFINES, and the resulting code can be a challenge to read and maintain.

**Fix** Use string arrays instead.

<b>Message</b>	"Warning: \$point, information lost, replacing with constant."
----------------	--

**Explanation** FIXED data type, it can be more difficult to maintain.

**Fix** Change the way your code relies on the FIXED data type.

<b>Message</b>	"Warning: Block move expression converted, but lost information due to repetition factor."
----------------	--

**Explanation** Literal values in block moves can be lost.

**Fix** Simplify the block move expression.

## 10.4 Significant Information Messages

<b>Message</b>	Your TAL2C license is invalid or missing.
----------------	---

**Explanation** Either TAL2C has found your license file (.license) but it is invalid or TAL2C cannot find your license file.

**Fix** If your license file is invalid, ensure that you are using the correct license file for your conversion project. Ensure that you have not changed the name of any file in the project, added any additional file(s) to the project, or made major changes to any file in the project since the license was generated. In any of these cases you may need to contact Gresham for a new license.

If TAL2C cannot find your license file, ensure that the correct file is on the workstation where you intend to run TAL2C in the same folder as tal2c.exe. If this is the first time you have run TAL2C, ensure that you have copied .license.dmo to .license so that you can convert the test and demonstration programs.

<b>Message</b>	Some defines have not been properly resolved. You must run TAL2C again to resolve them.
----------------	---

**Explanation** Your TAL source code contains DEFINES, and TAL2C was unable to convert them.

**Fix** If TAL source code contains DEFINES, TAL2C requires at least two passes over the source code to convert them.

If this message appears during the first pass, you must immediately run TAL2C again to attempt to resolve the DEFINES. You must not change the TAL source code before running TAL2C again. You should use the same TAL2C command that you used for the first pass.

If this message appears during the second pass, TAL2C is unable to resolve some DEFINES because they are unconvertible. You can either rework the original TAL code and run TAL2C again or change the C code.

## 11 What To Do If Things Go Wrong

Above all, don't panic! Your first step should be to consider carefully what has gone wrong, and this chapter provides some helpful hints to find out. If you cannot solve a problem using information in this chapter, you should not hesitate to contact Gresham for expert help. This chapter covers:

- Where to Find Helpful Information
- General Guidelines for Solving a Problem
- Information Required to Solve a Problem
- Sending Information to Gresham
- What Happens After You Send Information to Gresham
- Cleaning Up After an Abnormal Termination

### 11.1 Where to Find Helpful Information

Consider whether you can solve the problem by simply rereading the TAL2C documentation (including the `readme.txt` file). For example:

#### **Could it be a hardware or software problem (see Chapter 3, "Hardware/Software Requirements," in the *TAL2C Product Overview*)?**

- Do you have sufficient memory and hard disk space on your workstation?
- Are you using the correct version of Windows NT?
- Are you using an ANSI-compliant C compiler?

#### **Could it be an installation problem (see Chapter 2, "Installing TAL2C," and Chapter 3, "Getting Started")?**

- Have all files been correctly installed?
- Have you run the batch program, `install.bat` to rename files correctly?
- Have you set up the search path correctly?
- Do you have a valid license; that is, have you copied `.license.dmo` to `.license`?

#### **Could it be a configuration problem (see Chapter 4, "Configuring TAL2C")?**

- Is there a syntax error in the `.ctrans` configuration file (indicated by an error message)?
- Is there a logic error in the `.ctrans` file (for example, you omitted the name of a required TAL file)?

#### **Could it be an execution problem (see Chapter 5, "Executing TAL2C")?**

- Have you set up the search path correctly?
- Are you using the correct syntax to run TAL2C?

- Have you specified a nonexistent file name?
- Have you specified the wrong file?

**Could it be a license problem (see Chapter 6, "Licensing TAL2C")?**

- Do you have the correct type of license?
- Have you obtained a `.license` file from Gresham?
- Have you modified the `.license` file?
- Is the `.license` file in the correct location?
- Has the license expired?
- Does the license permit you to convert a certain file?
- Have you added TAL code to a file after obtaining a license to convert that file?

**Could it be a problem with TAL2C locating a file during conversion or some other file problem (see Chapter 3, "Getting Started," and Chapter 7, "TAL2C Usage Guidelines")?**

- Have you transferred all source files to the correct locations on your workstation?
- Have you transferred all source files in the correct mode (text mode)?
- Have you specified the folder for files that are included in by the TAL source file?

If you cannot find an answer in the documentation and you suspect that the problem may be related to the conversion process itself, follow the guidelines described in "General Guidelines for Solving a Problem".

## 11.2 General Guidelines for Solving a Problem

The following guidelines may help you to solve a problem if you suspect that the problem may be related to the conversion process itself.

Before continuing, however, you should confirm that the TAL source code being processed compiles without errors. This is always important, especially if you have not compiled the TAL code for some time. If there are compilation errors, these **must** be corrected before continuing.

If TAL2C is unable to convert a line of TAL code, it continues processing and displays an error message. The message indicates the file name, line number, and column number of the offending TAL source code line. TAL2C puts a representation of the offending TAL code in the C output code. In this case, you can either rework the original TAL code and run TAL2C again or change the C code.

If TAL2C converts a line of TAL code and you suspect the resulting C output code, you can either rework the original TAL code and run TAL2C again or change the C code.

In the unlikely event that TAL2C abends, it displays an error message. The message indicates the file name, line number, and column number that approximates the position of the TAL source code being processed when the abend occurred. You should contact us immediately to see if we can provide a solution to this problem (perhaps there is a later release of TAL2C that fixes the problem). We may ask you to provide us with certain information (see "Information Required to Solve a Problem").

### 11.3 Information Required to Solve a Problem

The information we may ask for to help solve a problem includes the following:

**Your contact details:**

- Your name
- Company name
- Postal address
- Email address
- Telephone number(s)
- Fax number

**Details of the software with which the problem occurred:**

- TAL2C version number and release date
- Microsoft Windows NT version
- Microsoft Windows NT service pack version

**Copy of the files being used when the problem occurred:**

- The TAL source code being converted (or, at least, a subset of the code that is sufficient to demonstrate the problem)
- C output code if the problem relates to the conversion process
- The `.ctrans` configuration file being used
- A log file to which you have redirected TAL2C messages, if available

Any other information that we think or you think is relevant to the problem.

### 11.4 Sending Information to Gresham

After you have gathered together all relevant problem files, we suggest that you use a compression utility (such as PKZIP or WinZip) to compress all the files into a single file. It is often faster and more convenient to provide us with a single compressed file rather than multiple files.

You can then send the single compressed file to us. If you use email (recommended), you can send the file as an attachment to an email message. If you plan to use email, you must ensure that you MIME-encode or uuencode the file so that is suitable for transmission through the Internet.

Alternatively you may be able to send us the single compressed file on CD, diskette(s), or on a (Tandem backup) magnetic tape.

See "How to Contact Gresham" for the options available to send information to Gresham.

### 11.5 What Happens After You Send Information to Gresham

After you send information to us to report a problem while using TAL2C, we will assign a unique reference number to the report.

You should use this reference number in all subsequent contact with us regarding the report.

After investigation, a member of our support staff will contact you to discuss the outcome.

## 11.6 Cleaning Up After an Abnormal Termination

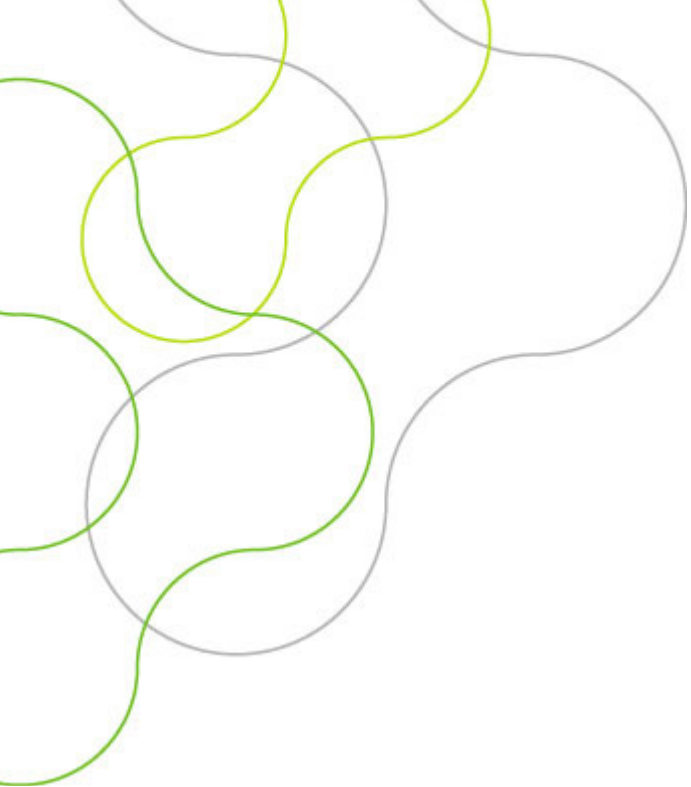
If TAL2C abends, you should contact us immediately and provide us with certain information (see "Information Required to Solve a Problem"). Additionally we recommend that you do not continue with your conversion project until you have ensured that we have all the information required to solve the problem.

**Note:** TAL2C always abends if it detects a file access error (for example, it is unable to locate a file during conversion).

You may also want to copy all relevant project files to another folder on your workstation until we have resolved the problem.

We may ask you run `cleanup.bat` in the folder with the TAL source code that caused the abend. This is a batch file to remove TAL2C intermediate files from the current folder. (Intermediate files are created during conversion, and include `.defines`, `.index_file`, and `REGION_` files.)

**Note:** Only run `cleanup.bat` if you intend to start a conversion project from the beginning or you are advised to do so by Gresham. You must run `cleanup.bat` more than once if you are converting TAL source code from more than one folder and want to clean up in each folder.



## About Gresham

Gresham Computing plc (LSE:GHT) specializes in the provision of real-time financial solutions to banks and corporates, and has a well-deserved reputation for technical excellence, reliability and a strong service culture. Our storage division helps the largest data users to better manage the unrelenting growth of data.

## Further information

For more information on how TAL2C can help your company visit

[www.gresham-computing.com](http://www.gresham-computing.com)  
or you can email us at  
[tal2c@gresham-computing.com](mailto:tal2c@gresham-computing.com)

Alternatively you can contact our offices directly.

Europe, Middle East and Africa

T +44 (0)20 7653 0200

Americas

T +1 416 620 6683

Asia Pacific

T +61 2 9955 7660

