



# **Interfacing C and TAL In The Tandem Environment**

Rod Norden

Technical Report 91.1  
April 1991

Part Number 22612



# Interfacing C and TAL In The Tandem Environment

**Rod Norden**

Software Development Education Group

## **Abstract**

Tandem Computers first offered a C compiler as a product in 1985. Since then, its use internally and among Tandem customers has grown dramatically. C is often used to write portable code and to import products from outside vendors into the Tandem environment.

The purpose of this paper is to provide information to development, support, and customers about several commonly asked questions:

- ANSI Standard C and the Tandem Implementation
- Interfacing C with TAL and Guardian
- Limitations on Writing NonStop Code in C



## 1.0 Introduction

The purpose of this paper is to provide information to development, support, and customers about several commonly asked questions:

- ANSI Standard C and the Tandem Implementation
- Interfacing C with TAL and Guardian
- Limitations on Writing NonStop Code in C

The information in this article is exclusively derived from a course (of the same title) that I prepared for the Software Development Education Group in the fall of 1989. The course has been taught within Tandem Software Development since February 1990. With the increasing use of C outside development, it is time to disseminate the contents of the class to a wider audience. Please note that this article also assumes a complete familiarity with the material provided in the *C Reference Manual* (PN 27359). All programming examples in this paper assume the default Large Model.

## 2.0 ANSI Standard C and the Tandem Implementation

Tandem has very nearly completed implementing the ANSI Standard. There are a few relatively minor exceptions currently (Release C30 with IPMs). Some ANSI features Tandem C has not yet implemented are const and volatile type qualifiers, signal/raise, and multibyte characters. A complete list follows in Appendix A.

It is recommended that you write C code with portability always in mind. Write only ANSI code whenever possible and keep it together in modules. Isolate all non-ANSI or Tandem specific code into separate modules which are clearly labelled as such, so only these segregated portions need to be re-written if the code must be ported to another machine. There are many good articles and texts describing how to write portable ANSI C, and I refer the reader to them for additional guidelines.

## 3.0 Interfacing C with TAL and Guardian

To understand how to interface C and TAL (and therefore Guardian procedures), one must understand the:

- lexical differences between C and TAL
- data type differences between C and TAL
- programming environments set up by the two compilers
- calling TAL from C and vice versa
- calling Guardian procedures from C
- sharing data between C and TAL

### 3.1 Language Differences Between C and TAL

Both C and TAL have independent sets of keywords. TAL allows the carat (^) within identifiers and C does not. Tandem C provides a mechanism (Interface Declarations) to handle this difference in mixed language procedure calls.

In a mixed language environment, data type compatibility is an important issue in parameter passing, function return types, and types of pointers to use for sharing data. Accessing data variables between C and TAL is possible only if the data to be shared has a compatible type. Known scalar type incompatibilities are:

- TAL has no numeric type compatible with C's **unsigned long**
- Only TAL FIXED(0) is compatible with C's **long long** (not FIXED(≠0))
- TAL's 16-bit INT is guaranteed to be compatible with C's **short** over all future Tandem hardware
- TAL's UNSIGNED(<n>) is not compatible with C's **unsigned short**. Regardless of the declaration in TAL, only the math operator in TAL determines whether signed or unsigned operations take place.

Another difference is that C is case sensitive and TAL is not. TAL's case insensitivity is implemented by upshifting all characters before compiling. Also note that Binder implements its case insensitivity by upshifting characters as well. Use MODE NOUPSHIFT in Binder if any reference is made to any C identifier in Binder commands.

### 3.2 Data Type Differences Between C and TAL

C provides arrays which are always zero indexed and which can be declared with multiple dimensions. TAL provides one dimensional arrays which can be indexed by declaring maximum and minimum values. In order for data in an array to be shared between C and TAL, the TAL array ought to be declared with zero as its initial index. The corresponding C array must have a single dimension.

Structures in C and TAL have one fundamental difference as well. All C structs and members which are structs themselves are word-aligned objects, i.e. they must begin on a word boundary. All TAL STRUCTs, whether declared directly or by referral to a template, are also word-aligned objects. Substructures within TAL Structures, declared by referral to a template, are word-aligned objects as well. In these cases, the TAL STRUCTs will map directly onto an equivalent C struct.

Here is the incompatibility. TAL substructures that are declared directly within their enclosing STRUCT are byte-aligned. This does not match C member struct alignment rules. If a TAL substructure of this type directly follows a field that ends on an odd-byte boundary and the substructure contains an initial byte-aligned field, sharing of data will be impossible without forcing realignment of

data within one language or the other. An example of this problem follows in Figure 1.

DDL understands and enforces structure alignment differences between C and TAL; therefore the user should generate all structure declarations (to be shared) using DDL. The resulting data structures are guaranteed to be compatible.

### 3.3 Programming Environments in C and TAL

An important difference between C and TAL is the fact that C has an extensive run-time library and TAL has none (other than Guardian). Currently, in a shared language environment, the language which has the '**main**' routine determines which run-time library is bound into the object file. Much of the power of C comes from the usefulness of the routines in its run-time library. Using C without its run-time library is not recommended -- one loses access to the memory management routines, the convenient string handling routines, and all I/O routines, among many others. It is the author's opinion that C without its run-time library is just not C (unless you have to write NonStop code in C which is discussed later).

A potential solution to this problem in a mixed language program is to have a dummy 'main' routine in the C code which calls the original TAL code MAIN procedure. TAL's original MAIN procedure will require some code changes and will need recompilation. First, the original TAL procedure will need its attribute MAIN removed. Second, the new TAL procedure cannot call the INITIALIZER as it is called by the C run-time library before control is given to the user's code. Although the C run-time library initialization routines also call ARMTRAP, it is possible for the new TAL procedure to call ARMTRAP again, but be aware that the C trap-handler will be disabled following a second call to ARMTRAP from TAL.

Please note that all information in the Guardian startup message sequence can be obtained in C using GETENV library procedure, including in and out files, default volume and subvolume, and TACL params. An exception exists for ASSIGN messages (other than that for file stderr), which cannot be read using GETENV. If it is absolutely necessary for TAL to call INITIALIZER or ARMTRAP or to read general ASSIGN messages, then C and TAL cannot *effectively* share code. An example follows in Figure 2.

In both memory models, C reserves the upper 32k of the User Data Segment for its run-time library data storage. Older programs written in TAL occasionally made use of this area with 16-bit word addresses in the days prior to release B00. The few TAL programs which use this area cannot run within an object file bound with a C 'main' routine.

TAL uses the standard Tandem hardware model of memory usage, and C provides two models, the Large Model using 32-bit addressing internally and the Small Model using only 16-bit addresses internally. All software development in C should use the Large Model (the default).

The C Large Memory Model stores all global aggregates (arrays and structures) and the heap in a single extended data segment (with SegID = 1024). This is equivalent to TAL's automatic allocation of an extended data segment to hold directly-declared extended data arrays and structures (also in SegID = 1024). The Binder stores eligible C and TAL data objects in this automatic extended segment according to Binder's standard named data block scheme. Note that it is possible in C to reference an aggregate with a 16-bit address using the non-ANSI storage class declaration *lowmem*, documented in the *C Reference Manual* and illustrated in Figures 5 and 6.

In a mixed environment, both C and TAL program modules are free to allocate and use additional extended data segments using Guardian's `ALLOCATESEGMENT` and `USESEGMENT` procedures. Users must be careful to restore SegID 1024 into use with `USESEGMENT` **before** C or TAL attempt any data accesses into their private segment (SegID 1024). Note also that Guardian provides the `MOVEX` procedure to allow data to be moved directly between extended data segments. An example using extended data segments in both languages follows in Figure 9.

### 3.4 Calling C Functions From TAL

This information applies to the case where the TAL component code contains a procedure with the attribute **MAIN**. Recall that since C does not have the 'main' function, C will not have access to its run-time library. This implies that the C functions called from TAL will have access to a relatively limited set of operations.

The C functions to be called from TAL must be declared external in the TAL code. C functions that return type void are represented as TAL procedures which return no value in the TAL module. C functions that return data types other than void are represented as typed TAL function procedures which return an equivalent data type value in the TAL module. A list of compatible data types is given in Section 18 of the *C Reference Manual*.

An example of calling C functions from TAL follows in Figure 3.

### 3.5 Calling TAL and Guardian OS Procedures From C

This information applies to the case where the C component code contains the '**main**' function, and therefore, has full access to its run-time library and its complete set of operations. This is the most often used and most powerful case of mixing the two languages.

An Interface Declaration for the TAL procedure to be called must be given in C prior to the call and replaces the typical external function declaration used in C. An Interface Declaration, initiated by the keyword 'tal', provides the C compiler with the additional data needed to interface correctly with procedures written in TAL. Specifically, it gives the C compiler four important pieces of information about the TAL procedure to be called.



- 1) attributes of the TAL procedure which affect how the parameters should be stacked for TAL
- 2) type of data returned from the TAL procedure or function
- 3) conversion of illegal TAL procedure names into valid C identifiers
- 4) types of parameters passed to TAL, if any

After giving the non-ANSI Interface Declaration, one uses standard C syntax to call any TAL procedure. The only exception occurs when calling TAL procedures with variable or extensible attributes in which case C allows TAL-style missing parameters in the C call (illustrated in Figure 6). See Section 18 of the *C Reference Manual* for complete Interface Declaration syntax.

Interface Declarations are provided for all Guardian OS procedures in the Tandem-exclusive <CEXTDECS> header file provided with the compiler. Note also that condition codes returned by some Guardian procedures are handled as return types and require the use of the Tandem-exclusive <TALH> header file provided. Note that condition codes are interpreted as Guardian function return values in C.

Some examples of Interface Declarations follow in Figure 4.

An example follows in Figures 5 and 6, showing the implementation of a simple requester-server program. It illustrates how to call Guardian procedures from C.

### 3.6 Sharing Data Between C and TAL and Vice Versa

Sharing of all types of data, whether scalars or aggregates, will be done via pointers initialized by passing addresses between languages. Some pointer differences between C and TAL are:

- A C pointer declaration defines the **exact** type the pointer can reference.
- A TAL pointer declaration **only** states whether the pointer contains a word or byte address, and is often initialized to a fixed location in memory.
- In both C and TAL, the programmer must explicitly initialize the value of the pointer variable.
- C pointers are **explicitly** dereferenced.
- TAL pointers are **implicitly** dereferenced.

In the Large Model, all C pointers are 32-bit extended pointers, whether or not they point to objects declared 'lowmem'. You must be careful when passing C pointers to TAL, since normal TAL pointers occupy only 16-bits. It is often easier to declare most TAL pointers to shared objects as 32-bit pointers so that you don't have to think about where the C compiler places data objects to be shared.

Unless a TAL procedure specifies that a parameter is an extended (.EXT) pointer, the C pointer you must pass to TAL must refer to an object in the User Data Segment. If the object in C is a global scalar or is declared local within a function, it is guaranteed to be in User Data. However, if the object in C is a global

aggregate, you must explicitly direct the compiler to store the aggregate object in User Data by including the non-ANSI keyword 'lowmem' in the object's storage class.

The recommended method to use when sharing data between C and TAL is simple. First, decide whether C or TAL will contain the data declaration that causes storage to be allocated. Whichever language you choose will 'own' the data. Suppose for example, you choose to let TAL 'own' the data. Then you will declare in C an uninitialized pointer (of the appropriate type) to the data to be shared by C. Next, you will declare a function in C which accepts an address (a pointer to the data to be shared) from TAL. This C function uses the address sent from TAL to initialize the pointer declared earlier in C to the shared data. Now C has access to the data 'owned' by TAL regardless of where C, TAL, and the Binder decide to allocate the shared data's storage on current or future Tandem hardware.

See the example in Figure 7 when TAL owns the data to be shared.

See the example in Figure 8 when C owns the data to be shared.

### 4.0 Writing NonStop Code in C

There are several properties of our implementation of C which must be considered when planning to write NonStop code in C. The traditional method of full-context checkpointing requires the primary process to CHECKPOINT **all** data that has changed to a non-busy backup process at appropriate intervals. The primary sends a control block or data to the backup and the backup just stores the data at the same addresses used by the primary process. Note that the memory state information (addresses) is sent with the data. This allows the backup to maintain a fully matching memory image with the primary process whenever new or updated information is sent in the checkpoint message. This method is called 'physical checkpointing', and is well suited to single-threaded processes.

In C, some run-time library routines maintain global data whose locations are unavailable to the programmer. Thus, one cannot include this data in checkpoints of context to the backup process since there is no way to find either where the data is or when it is changed. The location of the data is determined by the Binder and it varies from release to release.

Most of the C library routines that have global data are I/O functions, string manipulation functions, and memory management functions. A list of these routines for C30 is given in Appendix B. The set of routines varies from release to release and new routines are often added with each release, making the set difficult to document for users. It is therefore highly recommended that users do not develop code that depends on the locations of this data as they are guaranteed to change as the future development of the C compiler requires.

It is possible to write traditional full-context checkpointed NonStop code in C if the library routines maintaining private data are not invoked by the C program. Another option that has been used is for the programmer to provide their own custom version of the library routines that maintain hidden data. The programmer can then determine where all data is located and when it changes to supply it to Guardian's checkpointing procedures.

### 5.0 Summary

Tandem C provides several extensions to ANSI standard C that make it an effective language for writing applications which execute under the Guardian 90 Operating System. Tandem C allows access to several types of Tandem physical files. Tandem C is also well suited to porting applications implemented in C to the Tandem platform(s). Furthermore, a C interface to NonStop SQL is provided and documented in *NonStop SQL Programming Reference Manual for C* (PN 22967).

As long as the user keeps in mind the few simple rules discussed in this paper, interfacing C with TAL or Guardian will be a straightforward and rewarding programming effort.

## Figure 1

----- NON-EQUIVALENT C CODE -----

```
#pragma XMEM
struct rec2cNEQ
{
    short x;
    char a;
    struct {
        char b, c, d;
    } st;
    char e;
}
struct rec2cNEQ NotLikeTAL;
```

----- EQUIVALENT TAL CODE -----

```
STRUCT rec2t (*);
BEGIN
    INT x;
    STRING a;
    STRUCT st;
        BEGIN
            STRING b, c, d;
        END;
    STRING e;
END;

STRUCT .EXT match ( rec2t )
```

----- EQUIVALENT C CODE -----

```
#pragma XMEM
struct rec2cEQ
{
    short x;
    char a;
    char stb;
    char stc;
    char std;
    char e;
}

struct rec2cEQ JustLikeTAL;
```

## Figure 2

```
#pragma XMEM

#include <stdlibh> nolist

int main ( )
{
  char *env_parm1, *env_parm2, *env_parm3,
        *env_parm4, *env_parm5;

  env_parm1 = getenv("STDIN");
  if ( env_parm1 == NULL )      /* action */ ;
  env_parm2 = getenv("STDOUT");
  if ( env_parm2 == NULL )      /* action */ ;
  env_parm3 = getenv("STDERR");
  if ( env_parm3 == NULL )      /* action */ ;
  env_parm4 = getenv("DEFAULTS");
  if ( env_parm4 == NULL )      /* action */ ;
  env_parm5 = getenv("MYPARAM");
  if ( env_parm5 == NULL )      /* action */ ;

}      /* end main */

/* sample usages...
   tacl> assign stderr, $s.#rod
   tacl> param myparam mytextstring
*/
```

## Figure 3

----- EQUIVALENT C CODE -----

```
#pragma XMEM

short CFUNC ( char *s ) /* note uppercase CFUNC */
{
    /* to match TAL proc name */
    *s = 'A';
    s[2] = 'C';
    return 1;
} /* end CFUNC */
```

----- EQUIVALENT TAL CODE -----

```
INT status := 1;

STRING .EXT array [0:4] ! Note starting index 0 !

INT PROC cfunc ( a ); ! cfunc will be upshifted before !
    STRING .EXT a; ! C sees it. !
EXTERNAL;

PROC example1 MAIN;
BEGIN
    status := cfunc ( array );
END;
```

## Figure 4

### SYNTAX:

```
tal [variable]  [void]  identifier ["tal^only"] (param-types);
      or         or
      [extensible] [cc_status]                  (void)
              or
              [return-type [*]]
```

### EXAMPLES: from <CEXTDECS>

```
tal void DEBUG (void);
tal void DELAY (long);
tal cc_status POSITION (short, long);
tal short GIVE_BREAK = "GIVE^BREAK" (short *);
tal variable short ALLOCATESEGMENT (short, long, short *, short);
tal variable cc_status DEALLOCATESEGMENT (short, short);
tal variable cc_status READ (short, short *, short,
                             short *, long);
tal extensible cc_status READX (short, extptr char*, short,
                               short *, long);
```

## Figure 5

```
#pragma XMEM

#include <talh>      nolist
#include <fcntlh>    nolist
#include <stdioh>    nolist
#include <stdlibh>   nolist
#include <stringh>   nolist
#include <cextdecs ( OPEN,      \
                      WRITEREAD, \
                      FILEINFO,  \
                      CLOSE,     \
                      DEBUG )>   nolist

#define NOFILE      -1
#define MAXCNT      80
#define FILE_NAME   "$SRV"
#define FILE_NAME_SIZE 36

short main()
{
    lowmem short    ServName [12];
    short           ServNum;
    lowmem char      buffer [ MAXCNT ];
    short           wcnt = 0;
    short           rcnt = 0;
    short           error = 0;
    short           c_code = 0;

    fputs("C Requester started...\n", stdout);

    /* Convert external format filename to internal format */
    if (extfname_to_intfname(FILE_NAME, ServName) != 0)
        DEBUG(); /* error */

    c_code = OPEN (ServName, &ServNum);
    if ( c_code != CCE ) /* error */
    {
        FILEINFO(NOFILE, error);  DEBUG();
    };

    wcnt = stccpy ( buffer, "This is my request!", MAXCNT );

    c_code = WRITEREAD(ServNum, (short *) buffer,
                          MAXCNT, wcnt, &rcnt);
    if ( c_code != CCE ) /* error */
    {
        FILEINFO(ServNum, error);  DEBUG();
    };
}
```



```
printf("Server: Server answered \"%s\"\n", buffer );

c_code = CLOSE ( ServNum );
if ( c_code != CCE )
    exit ( EXIT_FAILURE );
}
```

## Figure 6

```
#pragma XMEM

#include <talh>      nolist
#include <fcntlh>    nolist
#include <stdioh>    nolist
#include <stdlibh>   nolist
#include <stringh>   nolist
#include <cextdecs ( OPEN,      \
                        READUPDATE, \
                        REPLY,      \
                        FILEINFO,   \
                        CLOSE,      \
                        DEBUG )>    nolist

#define NOFILE      -1
#define MAXCNT      80
#define FILE_NAME   "$RECEIVE"
#define RECVDEPTH   1

short main()
{
    lowmem short    RecvName [12];
    short           RecvNum;
    lowmem char     buffer [ MAXCNT ];
    short           wcnt = 0;
    short           rcnt = 0;
    short           error = 0;
    short           c_code = 0;

    fputs("C Server started...\n", stdout);

    /* Convert external format filename to internal format */
    if (extfname_to_intfname(FILE_NAME, ServName) != 0)
        DEBUG(); /* error */

    /* note TAL-style param syntax in next call*/

    c_code = OPEN (RecvName, &RecvNum, /*flags*/, RECVDEPTH );
    if ( c_code != CCE )
    {
        FILEINFO(NOFILE, error); /* error */
        DEBUG();
    };

    c_code = READUPDATE (RecvNum, (short *) buffer, MAXCNT, &rcnt );
    if ( c_code != CCE )
    {
        FILEINFO(ServNum, Error); /* error */
        DEBUG();
    };
}
```

```
printf("Server: Requester sent \"%s\"\n", buffer );

wcnt = stccpy ( buffer, "This is my reply!", MAXCNT );

c_code = REPLY ((short *) buffer, wcnt );
if ( c_code != CCE )
    exit ( EXIT_FAILURE );          /* call FILEINFO */

c_code = CLOSE ( RecvNum );
if ( c_code != CCE )
    exit ( EXIT_FAILURE );
}
```

## Figure 7

```
----- C CODE -----

#pragma XMEM

short *tal_int_ptr1;          /* ptr to TAL data */
char *tal_char_ptr2;

/* note caps in next line */
short INIT_C_PTRS_TO_TAL (short *tal_ptr1, char *tal_ptr2)
{
    tal_int_ptr1 = tal_ptr1;
    tal_char_ptr2 = tal_ptr2;
    return 1;
}

/* do work using initialized pointers */

----- TAL CODE -----

! data 'owned' by TAL !
STRUCT rec1 (*);
BEGIN
    INT x;
    STRING arr2[0:9];
END;

INT .EXT tal_arr[0:4];

STRUCT .EXT tal_struct (rec1);

INT status := -1;  ! use next proc as example of function proc !

INT PROC init_C_ptrs_to_TAL (tal_ptr1, tal_ptr2);
    INT .EXT tal_ptr1;
    STRING .EXT tal_ptr2;
EXTERNAL;

PROC TAL^name MAIN;
BEGIN
    status := init_C_ptrs_to_TAL (tal_arr, tal_struct.arr2);
    !.more work.!
END;
```

## Figure 8

```

----- C CODE -----

#pragma XMEM
#include <stdio.h> nolist

short arr[5];          /* data 'owned' by C */
char charr[5];         /* data 'owned' by C */

tal void INIT_TAL_PTRS_TO_C ( extptr short *, extptr char * );

tal void Legal_C_Name = "illegal^C^name^in^TAL" (void);

void example_func( int *x )
{
    printf("x before TAL = %d\n", x[2] );
    Legal_C_Name( );
    printf("x after  TAL = %d\n", x[2] );
}

main ()
{
    INIT_TAL_PTRS_TO_C ( &arr[0], &charr[0] );    /* init ptrs */
    /* test pointer values */
    arr[0] = 8;
    example_func( arr );
    arr[2] = 18;
    charr[2] = 'B';
}

----- TAL CODE -----

INT .EXT c_int_ptr;          ! ptr to C data !
STRING .EXT c_char_ptr;

PROC init_TAL_ptrs_to_C ( c_addr );    ! called from C !
    INT .EXT c_addr1;
    STRING .EXT c_addr2;
BEGIN
    @c_int_ptr := @c_addr1;
    @c_char_ptr := @c_addr2;
END;

PROC illegal^C^name^in^TAL;
BEGIN
    c_int_ptr[0] := 10;
    c_int_ptr[2] := 20;
    c_char_ptr[2] := "A";
END;

```

## Figure 9

```
----- TAL CODE -----

INT .EXT array[0:19];          ! bound into SegID 1024 !
INT .EXT arr_ptr;

?NOLIST
?SOURCE $SYSTEM.SYSTEM.EXTDECS(ALLOCATESEGMENT,USESEGMENT,DEBUG)
?LIST

PROC lose_seg;
BEGIN
    INT status := 0;
    INT old^seg := 0;
    INT new^seg := 100;
    INT(32) seg^len := %2000D;          ! 1024D !

    array[0] := 10;    ! do some work in SegID 1024 !

!   get a private segment for use by this TAL segment !

    status := ALLOCATESEGMENT( new^seg, seg^len );
    IF status <> 0 THEN CALL DEBUG;

    old^seg := USESEGMENT( new^seg );
    IF <> THEN CALL DEBUG;

!   use DEFINEPOOL, GETPOOL, etc. to retrieve a block in SegID 100 !

    arr_ptr := %2000000D;
    arr_ptr[2] := 10;    ! do some work in SegID 100 !

!   Don't forget to reset the Extended Segment (SegID=1024) that
!   C will be expecting when you return to it...
!   The C code will run until you either access an invalid address
!   or you need a library routine which would be inaccessible;
!   you will also get the wrong data from any valid address
!   in SegID 100.

    old^seg := USESEGMENT( old^seg );    ! Now SegID 1024 will be !
    IF <> THEN CALL DEBUG;                ! accessible upon return !

END;
```

```
----- C CODE -----

#pragma XMEM

short numarr[10];          /* global aggregates */
char sarr[10];
char *s;

tal void LOSE_SEG (void);  /* Interface Decl--note UC proc */

main ()
{
    s = &sarr[0];
    *s = 'A';
    numarr[0] = 10;

    LOSE_SEG();            /* next 2 stmts depend on SegID 1024 */
                           /* being restored after call to TAL */
    sarr[1] = *s;
    numarr[1] = numarr[0] + 5;
}
```

### APPENDIX A:

This Appendix details the ways in which the current C compiler is not ANSI compatible. This list assumes familiarity with C and the ANSI C standard. The section numbers used are those of the standard document.

For the purpose of this paper, the "current C compiler" is the last C30 Class A IPM (T9255AAR) which is in the field as this was written (March 1991). This document includes Tandem's plan for implementing the missing features.

#### Section

2.1.1.2: token continuation with \newline

What: The lexical analyzer should allow tokens to be split across a line boundary. For example:

```
in
t i; /* 't' would have to be in column 1 */
```

2.2.1.1: trigraph sequences

What: ANSI defines a set of alternative three character sequences for computers whose character set is not rich enough to support C's reserved tokens (such as curly brace).

2.2.1.2: multibyte characters

What: The standard defines a set of functions and semantics for multibyte character constants and strings.

2.2.4.2: header constants

What: The standard defines a set of constants whose machine specific values must be supplied in standard header (interface) files. The constants Tandem C does not yet define are:

```
MB_LEN_MAX, LDBL_MANT_DIG, LDBL_DIG, FLT_MIN_EXP, DBL_MIN_EXP,
LDBL_MIN_EXP, LDBL_MIN_10_EXP, FLT_MAX_EXP, DBL_MAX_EXP,
LDBL_MAX_EXP, LDBL_MAX_10_EXP, LDBL_MAX, LDBL_EPSILON, LDBL_MIN
```

3.1.3.4: L prefix to character constant to indicate multibyte

What: Support for multibyte character constants (that is, character constants where a single char takes multiple bytes, not two character constants). An example is a constant for a Kanji character.

3.3.2.2: allow pf() for (\*pf()) and (\*f()) for f()

What: The standard says that anywhere a function designator is expected, a function pointer can be supplied and visa versa. For example, ANSI allows a function 'f' to be called as any of the following:

```
f();
(*f());
(*****f());
```

conversely, using a pointer to a pointer to a pointer to a function 'pppf', under ANSI, the user can call the function any of the following ways:

```
(***pf());
(**pf());
(*****pf());
pf();
```



- 3.4: disallow casting non-arithmetic constants, e.g., address constants to arithmetic constants  
What: The standard does not support casting non-arithmetic constants (address constants) to arithmetic constants. Tandem C will implement this change as a warning so as not to break existing code.
- 3.5.2: type specifiers and other declaration specifiers in any order  
What: The standard specifies that type specifiers and storage class modifiers can come in any order. For example:  
    extern int a;  
could be written  
    int extern a;  
In our current implementation, the storage class modifier must precede the type.
- 3.5.3: type qualifiers: const, volatile  
What: The standard defines two type qualifiers "const" and "volatile". Tandem C does not yet support these.
- 3.6.4.2: allow switch statements with no cases  
What: The standard specifies that a switch statement with no cases is legal. Tandem C currently requires at least one case.
- 3.8: #elif  
What: An "else if" for multiple #if conditional compilation statements.
- 3.8.8: \_\_STDC\_\_  
What: The standard specifies that the token \_\_STDC\_\_ will be set to 1 in an ANSI standard compiler. By definition, this will be the last feature Tandem C implements.
- 4.1.3: The standard values for errno must be positive  
What: The ANSI standard specifies that all values of errno must be positive. Tandem C currently sets negative values for non-Guardian errors.
- 4.1.5: size\_t in the large model should be unsigned long offsetof() should expand to an integral constant expression; it must work as an initializer, etc.  
What: Size\_t should be a unsigned long in the large model, Tandem C currently implements it as a long. offsetof should be an integral constant - should be usable in global initializations.
- 4.1.6: library functions can be declared and if they return an int they can be called without a declaration the functions in ctype.h can only be macros if they evaluate their arguments only once  
What: any library function which is not var arg and returns int should be callable without a prototype in scope. This means that Tandem C needs entry points with the C name (not that TAL alias). Also, the functions in ctype.h should be macros only if the evaluate their arguments exactly once.
- 4.2.1.1: assert() calls abort()  
What: The assert macro should be implemented to call abort when the assertion fails.
- 4.4: locale.h  
What: This set of ANSI defined functions allows users to pick from a set of implementation-defined "locales". A locale defines various things about the current environment such as the currency symbol, decimal symbol, date format, etc. It is up to us which and how many

## Interfacing C and TAL in the Tandem Environment

---

locales Tandem C wants to support. The functions to be supported include `setlocale()` and `localeconv()`.

4.7: `signal.h`

What: Support for ANSI asynchronous exception handling (e.g., `signal` and `raise`).

4.9.2: text files must allow lines containing a minimum of 254 characters; edit files allow only 239 characters

What: ANSI specifies that TEXT streams support a minimum of 254 characters per physical line. The EDIT file format (our default TEXT stream file format) supports a maximum length of 239. To meet the ANSI requirement, Tandem C would either need to extend the limit on EDIT files (not practical) or add an option by which TEXT files are supported as odd-unstructured streams.

4.9.5.6: `setvbuf()`

What: Allows user to set size of buffering for I/O streams. A full implementation implies unbuffered streams.

4.9.9: `fgetpos()` & `fsetpos` functions and `fpos_t`, `_IOFBF`, `_IOLBF`, `_IONBF` macros

What: These functions support getting and setting the current position within files. Similar to `ftell` and `fseek`, they work on larger files and text files.

4.10.4.1: `abort()`

What: Causes an abnormal `abend` (like `exit()` with a non-zero value) unless `SIGABRT` is being handled and the signal handler does not return.

4.10.4.5: `system()`

What: Function takes a string argument. That argument is passed to the system's command interpreter which "executes" it.

4.10.7: `mblen()`, `mbtowc()`, `wctomb()`, `mbstowcs()`, `wcstombs()`

What: RTL functions to support multibyte character sets.

4.11.4: `strcoll()`, `strxfrm()`

What: collate and transformation functions for multibyte character strings.

4.12: ANSI time functions: `clock()`, `difftime()`, `mktime()`, `time()`, `asctime()`, `ctime()`, `gmtime()`, `localtime()`

What: These functions perform various functions on time (e.g., return the current time in various forms, subtract two times, etc.).

The locale and multibyte items affects the compiler and some of the other library functions in several ways other than the specific ways mentioned above. For example, having wide character strings means that string initializers must be allowed where each character in the string is 2 bytes instead of 1 and multibyte characters must be processed in comments. The locale items affects the behavior of some of the library functions such as `strcoll()`.

Any non-standard keyword, identifier, function, etc. that Tandem C has in its compiler and header files will have to be removed or disabled while in ANSI mode, e.g., `tal`, `trap_overflows`, etc. Division by a constant zero should not be a compile-time error. It should get a trap at run-time.


### APPENDIX B:

This Appendix details the C run-time library restrictions which must be adhered to in a passive NonStop C program. As discussed separately, these operations are restricted because the functions mentioned use global, static memory to keep state information between calls. The user program does not know where this data is or when it is changed and, therefore, has no way to checkpoint it. I would like to thank John Hausman for its content.

- You cannot call any I/O routines in the C library. However, you can call `sprintf/sscanf` (which look like I/O but are really string formatting and string parsing routines).
- You must use Guardian calls to do I/O.
- You cannot call C memory management routines (e.g., `malloc`, `calloc`, `free`).
- You cannot depend on the value of `errno` across checkpoint boundaries, unless it is checkpointed.
- You cannot access environment information (that is, call the `getenv` function or access the arguments to `main`). If you need this information, move it to a user global and checkpoint that.
- You cannot depend on the sequence of random numbers returned by `rand`.
- The functions `strtok`, `tmpnam` and `atexit` must not be called.
- You must compile with the `NOCHECK` directive. The `CHECK` option (on by default), causes the compiler and library to perform some assertions (e.g., asserting that parameters to library functions have reasonable values). Errors are reported via `fprintf` to `stderr`, so the `check` directive must not be used in a NonStop C program.





Distributed by  
 **TANDEM**  
Corporate Information Center  
10400 N. Tantau Ave., LOC 248-07  
Cupertino, CA 95014-0708