Assembler

There are some things you cannot do in C language that you can do in assembler. An example of this is modifying protected core - there is no glmod( ) function allowing you to do this.

As the TPF ISO-C environment continues to become more robust, the need for assembler routines is diminishing. for example, GI's resource checking macro (REUTC) does not have a C version either so earlier ISO-C applications had to code that in assembler also. IBM now provides a LODIC/lodic() facility for resource checking, so there is not longer a need for assembler calls to REUTC.

In any case, there will be times when you legitimately need to use assembler within your library or DLM. You do this by linking an assembler component in your DLM or library. The MCF record type will be 'W' just like other components, but the language will be 'A' for assembler instead of 'B' for C.

A good reference is "Coding Library Functions in Assembler" in the *TPF Application Programming* manual on the BookManager Galileo ISO-C bookshelf. Though the title implies it deals with library functions, the same information holds true for assembler components of DLMs.

If you want to look at some examples, there are two examples specific to modifying protected core:

- the Links group first created component VLF2, part of DLM VLU0, and
- Client File later created component PFUCR which is a part of DLM PF07.

There are a few routines in the Client File assembler components to handle other things that could not be done with C - like the UALOC. So they set up the first parameter to indicate which assembler functionality they needed.

Make sure you follow these guidelines:

- The assembler segment needs to have TPFISOC=YES on the BEGIN statement.
- The assembler segment probably needs to have TMSPC/TMSEC wrapped around its code. (See TMSPC/TMSEC in the *TPF General Macros* on the BookManager TPF bookshelf.) These macros emulate the prolog and epilog code normally generated by the C compiler around a C function. Most of the time you will want to code these macros. You do NOT need these macros if you do not need registers saved and restored before/after your code and otherwise have no need for a stack. We know of one example where these macros were not included. It was a component that merely returned the value of a Store Clock instruction. This is not a recommended practice.
- The assembler component does not have to be in the allocator and its name can be up to 6 characters long (plus 2 characters for version equals 8 character object member names in the objlibs).
- The assembler component is processed by the high level assembler with no changes to its options. Use PROASM with the same environment you use for any assembler segment on the system for which you are assembling.
- The C source module that calls the assembler routine needs to have access to a prototype for the assembler routine.
- The build script must include the assembler component in the list of components.
- If the assembler component calls any SVCs, you have to save and reload R8 across the macro call (see TMSPC example in General Macros manual).
- If you have an internal DSECT in your assembler component (not that this is a recommended practice…) you return to the control section with your program name instead of the IS you normally use in a TPF segment with a BEGIN. This because the TPFISOC=YES on the BEGIN generates a name CSECT instead of a IS START 0. Here's an example of the proper code for DSECT within an assembler component:

```
BEGIN NAME=MYCOMP,VERSION=TM,TPFISOC=YES
```

```
WORKAREA DSECT
BYTE     DS C
MYCOMP   CSECT
```

Galileo TPF C Coding Guidelines & Standards

Purpose

The TPF environment will adhere to the C guidelines and standards as set forth in *C Programming Guidelines, Second Edition for Standard C*, by Thomas Plum. This document is intended as an addendum to that publication for the purpose of containing Galileo International's exceptions, additions, and clarifications as needed to support the TPF environment.
The numbered section headers in this document refer to the headers in the Plum guideline book.

Chapter 1 DATA AND VARIABLES

- 1.1 lexical rules for variables
- 1.3 standard defined-types
- 1.4 constants
- 1.11 structures

Chapter 3 CONTROL statements

- 3.1 control structure

Chapter 4 FUNCTIONS AND OTHER MODULES

- 4.1 function style
- 4.2 #include <xxx.h > versus #include "xxx.h"
- 4.4 location of includes
- 4.5 standard compile-time flags
- 4.7 no initializations in headers
- 4.11 on the use of 'low-level I/O functions'
- 4.12 regarding binding time
- 4.19 on local standard headers

Chapter 5 GENERAL STANDARDS

- 5.2 Suggested Use of Comments
- 5.3 Regarding makefiles

Chapter 6 NOTES ON STANDARD C

Chapter 1 DATA AND VARIABLES

1.1 lexical rules for variables

Longnames

The IBM mainframe C compilers provide a LONGNAME compiler option which allows for mixed case names up to 255 characters in length. Even though it supports up to 255 characters, we should stay within the 31-character limit recommended by Plum. If LONGNAME is used, the prelinker must be used before the linkage editor - the prelinker will truncate/create unique 8 character uppercase names for external references. The supported development and move to production processes (PROASM and AMP) both specify the LONGNAME option and the link edit processes of CBLD and AMP have the prelink step.

Long Library Function Names

Library function names either have to be kept to 8 character unique names or mapped to 8 character unique names using a #define statement. This is a restriction imposed on us by TPF's use of library function stubs. At link/edit time the library calls are resolved using library function stubs. The stubs are stored in the MVS partitioned datasets, PD.TPF41.CLIB and TD.TPF41.CLIB, whose members' names must be uppercase with an 8 character maximum length.
In order to use longnames for library functions you must include a #define statement that maps the longname to a shortened name (8 character, upper case). The eight-character name must be unique from all other library functions. To verify its uniqueness, search PD.TPF41.CLIB and TD.TPF41.CLIB. You can use the LIB exec to do by typing the name that you want to use in the name field and typing PD.TPF41.CLIB or TD.TPF41.CLIB in the library name field. IBM has reserved the use of double underscores (and double @) for their names.
Here is an excerpt from the Client File header file pfplib.h that contains all of the defines for the Client File library functions.

```
#pragma nomargins nosequence
#ifndef pfplib
#define pfplib
/****************************************************************************/
/* Header File Name : PFPLIB.H                                              */
/* Header File Type : TPF                                                   */
/* Description : Header File control for Client File Library                */
/* functions                                                                */
/*                                                                          */
/****************************************************************************/
#define proc_build_address_imsg          PRADIM
#define proc_build_air_imsg              PRAIIM
#define proc_build_assoc_imsg            PRASIM
#define proc_build_elect_tick_imsg       PRETIM
#define proc_build_man_ssr_imsg          PRMSIM
#define proc_build_mileage_imsg          PRMIIM
```

```
#define proc_build_name_imsg            PRNAIM
#define proc_build_notepad_imsg         PRNOIM
#define proc_build_osi_imsg             PROSIM
#define proc_build_phone_imsg           PRPHIM
#define proc_build_preformat_imsg       PRPFIM
#define proc_build_pax_info_imsg        PRPIIM
#define proc_build_prog_ssr_imsg        PRPRIM
#define proc_build_review_imsg          PRREIM
#define proc_build_pax_reporting_imsg   PRRPIM
#define proc_build_seat_imsg            PRSEIM
#define proc_build_vendor_imsg          PRVEIM
#define proc_build_ticketing_imsg       PRTAIM
#define proc_build_tick_rmks_imsg       PRTRIM

    #endif
```

Name Collisions

Another issue with names of variables and functions is the name space, or scope, in which the names need to be unique. For example, if you code a local function called find_record_to_be_updated( ) the prelinker will truncate it to FIND@REC (underscores are changed into @). Then if you try to access the find_record( ) TPF library function, your link edit will resolve that call to your local FIND@REC!
Any externally known function must have a unique name or resolve to one or collisions WILL occur. The same is true for global data.

C++ Keywords

Do not use C++ keywords or the keywords that will likely be reserved for C++ for names. The C++ keywords are:

asm	catch	class	delete	friend	inline
new	operator	private	protected	public	template
this	throw	try	virtual		

Future versions of the C++ compiler may reserve the following keywords, so you should avoid using them in your applications:

```
bool     const_cast  dynamic_cast      explicit     false
mutable  namespace   reinterpret_cast  static_cast  true
typeid   typename    using
```

1.3 standard defined-types

Use explicit long or short when declaring integer variables. This will clarify to the user the size (value range) of the variable and avoid confusion caused by different platform implementations of short and long. Defining bit fields is an allowable exception since

bits must be defined as either int, unsigned, or signed. It is recommended that you use unsigned for bit fields.
See header file gitpf.h for standard defined types UCHAR, USHORT, ULONG, etc.

1.4 constants

Header file gitpf.h contains definitions for some constants including the following:

```
#define ON    1
#define OFF   0
#define TRUE  1
#define FALSE 0
#define YES   1
#define NO    0
```

1.11 structures

Galileo Macros to Use with Structures

See header file gitpf.h for macros to use in determining the offset of a structure member from another member, the length (in number of bytes) from one structure member to another, etc..

Structures and Assembler Data Macros

All data structures which have a corresponding assembler DSECT defined by their own data macro shall reside in their own header files. These header files shall be named by suffixing the name of the assembler data macro with "C". Since there is currently no MCF record tie between a DSECT and its structure, include a comment at the top of each cross-referencing the other. Also include a comment that indicates how this structure was created.

Padding

Gaps may be left in memory between elements of a structure to align elements on their natural boundaries. Eliminate the padding of bytes within a structure by using the #pragma pack preprocessor directive before structure definitions. This is particularly important when creating a C structure whose fields must be at the same displacement as an assembler DSECT.
The #pragma pack directive should be used instead of the _Packed keyword. For a complete explanation of this directive, see How to Pack.

Use of the #pragma pack(1) directive eliminates the complications involved in the use of _Packed keyword. If you are working with existing code that uses the _Packed keyword, you should be aware of these complications:

- If the _Packed keyword is used, all nested structures and unions within a structure need to also use the "_Packed" keyword or they will default to unpacked. This is one of the complications of the _Packed keyword, that is
- The use of the "_Packed" keyword on a structure definition does not have any effect on subsequent declarations of pointers to that structure or variable instances of that structure. Each of the pointers or variable declarations must also include the keyword "_Packed". In order to prevent errors in the declarations

for a packed structure, a typedef that includes the keyword "_Packed" should be used.

Here are two methods for using a typedef for a structure that uses the _Packed keyword:

```
_Packed struct ww0ww
{
   UCHAR xxxx;
   long yyy;
   UCHAR cc[3];
   short vvv;
};
typedef _Packed struct ww0ww WW0WW;
```

```
typedef _Packed struct ww0ww
{
   UCHAR xxxx;
   long yyy;
   UCHAR cc[3];
   short vvv;
} WW0WW;
```

Accounting for All Bytes

Ensure all bytes are accounted for in a structure by using an unnamed bit declaration and a comment. There should not be any displacements unaccounted for due to slack bytes or reserved space.

```
struct example
{
   long day_value;   /* field one - 4 bytes, fullword alignment */
   short day_score;  /* field two - 2 bytes, halfword alignment */
   int :16           /* spare, alignment purposes */
   long day_total;   /* field three - 4 bytes, fullword alignment */
    };
```

1.14 Unions

TPFDF LRECS

TPFDF uses unions to define the different LREC layouts. The following example illustrates an acceptable method of doing so:

```
#pragma pack(1)
struct gr861v
{
   short gr86siz; /* LREC size */
   unsigned char gr86key; /* primary key */
   union
   {
      struct gr86incl /* logical record type x"60" */
      {
         short gr86irl; /* rule number */
         char gr86icx; /* context of this rule */
      } K60;

      struct gr86cntl /* logical record type x"70" */
      {
         ...
      } K70;

      struct gr86cond /* logical record type x"80" */
      {
         ...
      } K80;
   } lrec;
};

#define GR86IRL lrec.K60.gr86irl
#pragma pack()
```

If the database being manipulated is truly a TPFDF database, use dft_siz (in gitpf.h) and dft_key (in c$cdfapi.h) as the data types for the size and key fields.

Chapter 3 CONTROL statements

3.1 control structure

Control Statement Bracing Styles

Of the bracing styles shown in the Plum guidelines book, the exdented style (#2) is preferred. Use fully braced blocks - if you put braces around one half of an "if" block, put braces around the "else" block also:

```
if (a == b)
{
   amort_loan(a);
```

```
   ++qualified;
}
else
{
   ++nerrs;
}
```

This makes it less likely for the logic to be distorted when adding a second statement into the else code.

Use of Columns 72-80

Columns 72-80 are assumed to contain sequence numbers and are ignored by the compiler. In order to use these columns in a source file, the first statement in the file must be:

```
#pragma nomargins nosequence
```

Chapter 4 FUNCTIONS AND OTHER MODULES

4.1 function style

Function Arguments Bracing Style

Regarding bracing styles, the option shown in the Alternatives section of the Plum guidelines is the preferred style if all function arguments fit on one line:

```
TYPE func(TYPE1 a1, TYPE2 a2)
{
_
}
```

A modified exdented style is otherwise preferred:

```
TYPE func(TYPE1 a1,         /* comment describing a1 */
        TYPE2 a2)         /* comment describing a2 */
{
_
}
```

4.2 #include <xxx.h > versus #include "xxx.h"

Use double quotes for anything written by GI, angle brackets for IBM-provided headers. The angle brackets specify system include files, and double quotation marks specify user include files.
When compiling against our production headers libraries (PD.TPF41.CMACLIB and RUA.PROD.CHEADER) which are PDS's (partitioned datasets) the difference in syntax has no effect. However, when compiling using header files in HFS directories, the double quotes and angle brackets have two different effects.

If you enclose the file name in double quotes, the preprocessor searches the directories or libraries that contain the user source files. It then searches a standard or specified sequence of places, until it finds the specified file. If you enclose the file name in the angle brackets, the preprocessor searches only the standard or specified places for the specified file.

See the C/C++ User's Guide in the OS/390 C/C++ Optional feature bookshelf in BookManager for a complete discussion of this topic.

Note: The names of header files should be in lower case. Please discontinue the practice of using mixed case or upper case. For example:

```
#include "giTPF.h"
```

should be changed to:

```
#include "gitpf.h"
```

4.4 location of includes

Order of Header Files

It is generally accepted practice to sequence include files starting at system level and finishing at application-specific headers to aid readability. If this is done, the application header will take the compiler error if a value in a system header is redefined in an application header.
The suggested order of include files for the TPF environment is:

```
#include "gifirst.h"    /* Galileo first header file */
                        /* for PC compile compatibility */

#include <stdio.h>      /* IBM-provided ANSI C header files */
#include <stdlib.h>
#include <string.h>

#include <tpfeq.h>      /* IBM-provided TPF specific header files */
#include <tpfapi.h>
#include <tpfio.h>

#include "gitpf.h"      /* Galileo system level definitions */
                        /* like TRUE, FALSE _*/

#include "fmsgc.h"      /* Application unique header files */
#include "wa0aac.h"
```

4.5 standard compile-time flags

System Identification

The compile time flags of APO_CORE and PRE_CORE can be used to control the header file concatenation. APO_CORE will call the Apollo header datasets, PRE_CORE

will call the Galileo header datasets followed by the Apollo header datasets. It is important to be sure that you code for the system id, or ids, valid for your code as well as allowing for expansion of future ids.

```
#ifdef PRE_CORE
_
#elif APO_CORE
_
#else
   #error "HOST SYSTEM ID NOT DEFINED"
#endif
```

Mainframe Compiler

To determine if you are compiling on the mainframe, the following check can be used:

```
#ifdef __COMPILER_VER__
  I am on the mainframe
#else
_
#endif
```

4.7 no initializations in headers

OPEN ISSUES: What then are acceptable solutions to sharing small tables of data between functions? How could you define that table if you use it as read only so it stays in that object file rather than incurring the overhead of being moved into a static block?

4.11 on the use of 'low-level I/O functions'

An example of a comparable guideline for TPF would be to use find_record( ) instead of fiwhc( ).

4.12 regarding binding time

See note in 4.5 for compile-time binding

4.19 on local standard headers

See note in 4.4 regarding the placement of gifirst.h and gitpf.h

Chapter 5 GENERAL STANDARDS

5.2 Suggested Use of Comments

A program fragment that demonstrates the suggested use of comments can be viewed by clicking on the following link.

COMMENTS EXAMPLE

C Component Shell

You can download a file that contains the comment blocks shown in the COMMENTS EXAMPLE above. To download:

1. Click on one of the two shell links below.
2. Select the FILE drop down menu.
3. Select SAVE AS.
4. In the SAVE AS dialog box choose a location and name for the file.
5. Click on SAVE.

72 COLUMN SHELL	80 COLUMN SHELL

The shells are also available on CMS using the CCOM exec.

5.3 Regarding makefiles

The equivalent in TPF is a Build Script which is used to tell what objects are to be linked together to create a DLM or library.

Chapter 6 NOTES ON STANDARD C

6.3 enums

In the IBM mainframe compiler there is a CHECKOUT(ENUM) option which does give you this enum checking. This compiler option is specified by PROASM.
Special care should be taken when using enums in structures because the size of a field declared as one of these types varies according to the maximum value. For example if the enum was enum small{first, second, third} then it would be one byte. But if, over time, the values got beyond 256, it would start to take up two bytes. Therefore a maximum value should be coded for enums within structures as shown in this example:

```
typedef enum cf_did {
  NO_DID,
  DID_ITIN,                 /* Itinerary */
  DID_ADDR,                   /* Addresses */
  DID_RMKS,                   /* Remarks */
  MAX_DID_VALUE=4095
} CF_DID;
```

6.4 preprocessor commands

#error

Using #error in a header file is a good way to update a 'C$' header file that has been renamed. The next time a component is compiled calling the C$ header file, the compile will fail with the text placed in the #error

#error Do not use c$wa0aa.h - please use wa0aac.h - Doug Eulberg x5965

#define vs. #pragma map

Use a #define statement to give a more user-friendly alias to an external reference. In the past we have used #pragma map statements for the same purpose. The #define statement is preferred over the #pragma since #pragmas are not portable. As an example:

```
#define output_via_fmsg FMSG
```

is preferred over

```
#pragma map(output_via_fmsg, "FMSG")
```

6.18 and beyond

Descriptions and prototypes for the IBM-provided TPF functions can be found in the IBM publication *C Language Support Reference Summary* . This booklet is given to students in the Technical Training C series. Also available in BookManager is the more complete *C Language Support User's Guide* , which contains information on all IBM-provided functions

Prototypes for Galileo libraries ADL0-9 can be found in gitpf.h.

Chapter 7 APPENDICES

7.2 Bibliography

For more information see the ISO-C page on the Galileo Intranet. Included there is detailed information on the move to production process, instructions on using PC-lint, a bibliography for further reading, instructions on how to find relevant IBM manuals in BookManager, a copy of this document, copies of ISOC Mailbox e-mails, and other information.

The address for that page is:
http://home.galileo.com/isoc/index.htm

7.3 Galileo International Software copyright statement

All Galileo International software must be copyrighted. This covers all software, including host and micro and products and enhancements. This includes both object and source code.

A copyright notice should be added at the beginning of the program. This can be accomplished by adding a preprocessor directive to your source code:

```
#pragma comment\
 (copyright,"(c) 1998 Galileo International. All Rights Reserved.")
```

For programs that are developed by Galileo International and do not have third-party licensed code imbedded in them, the copyright notice should read as follows:

"(c) 199x Galileo International. All Rights Reserved."

For programs that are enhancements to and imbedded in licensed third-party software (and are not separate programs), the copyright notice should read as follows:

"Portions of this software are copyrighted by Galileo International.
(c) 199x Galileo International. All Rights Reserved."

In each case, the 199x should be replaced with the year of first publication (i.e. year of first distribution). Where we are enhancing or revising code previously developed, the 199x should be replaced with the year of first publication of the base program and the year (or years) of first publication of the enhancement or revision. For example:

"(c) 1992-1995 Galileo International. All Rights Reserved."

All software developed by Development should be copyrighted (as described above) in the name of "Galileo International ". This includes software developed in Swindon.

These guidelines are effective for all software.

In the event of any questions, please contact the Legal Department.

CHAPTER 1

Introduction to the Course

This chapter provides you with an opportunity to review the course description, as well as the course procedures. Please do so by clicking on the appropriate links on the left side of the screen.

The Coding C for TPF Course Description

Purpose of the Course

This is an introductory course on applying the C programming language (ISO C) in a TPF environment.

Course Objectives

After completing the course, the participant, with the aid of the course materials, will able to do the following:

- Design, code, compile, and test C language programs in a TPF system, following recommended guidelines.
- Manage storage consistent with efficient resource usage.
- Map C structures to TPF DSECTs.
- Interpret TPF C listings to determine program functionality
- Use TPF C functions to control program processing paths
- Use TPF C functions to input and output data.
- Use TPF C functions to get and release working storage
- Use TPF C functions to access and update file records.
- Use TPF C functions to provide error handling information
- Use TPF C functions to access TPF globals.
- Code the interfaces between C and Assembler programs
- Analyze TPF C dumps to interpret C code and storage areas

Course Length

The student will have nine days in which to complete this course.

Audience

This course is an appropriate introduction for someone who will write C programs in TPF.

Course Prerequisites

Course pre-requisites are the following:

- A working knowledge of the VM/CMS operating system.
- A working knowledge of the TASTE test environment

- A working knowledge of the TPF Assembler programming environment
- A working knowledge of the ANSI C programming language

Methodology

This course is conducted in an instructor facilitated, self-paced manner. It consists of a number of chapters (lessons) which teach all of the skills associated with introducing C coding in TPF. Each chapter consists of material to be read, and a skill check (criterion test). After reviewing the material and feeling comfortable with the learned skill, the students will complete the skill check. There are few, if any, lectures with this approach. Upon completion of the skill check, the student will review their skill check answers with a course manager (facilitator or teacher). The course manager is responsible for insuring that the student has mastered the skill in that chapter. Once the course manager is confident that the student has mastered the new skill, the course manager signs the student off on the chapter. Each participant must meet the skill criterion in a particular chapter before proceeding to the next.

For the student, the major advantage of this type of training is that the student can proceed at their own pace. This approach is an advantage to both the quick learner and the slow learner. The student who is learning the material at a fast pace doesn't have to wait until the next scheduled lecture to learn more. Instead, they can proceed to the next chapter whenever they've successfully completed the previous chapter's skill check. It may turn out that they complete the course several days sooner than a slower student. Likewise, the student who is learning at a slower pace will not be forced to move ahead when the next lecture takes place, but instead they will move forward only when they have acquired the previous skill. In most technical courses, the skills build upon each other, and it makes little sense to move on to the second skill when you haven't mastered the first.

The course managers are always available to assist the student and to answer any questions. If it becomes apparent that a student is confused about a particular topic in the course (either due to the fact that the student is having trouble with a skill check, or if the student raises questions with a course manager), the student will receive a one-on-one explanation of the topic from the course manager. This "personal lecture" will have the advantage of being one-on-one, and of being taylored to the specific needs of that student. This approach allows the teacher to more closely monitor the progress of each student.

The Coding C for TPF Course Procedures

How to Begin

The course will begin with course manager and participant introductions, a discussion of the course and expectations of everyone involved in the course, and a presentation regarding the logistics of the course. After that, you will begin working your way through the course modules in a self-directed and self-paced fashion. That is, you may proceed at your own pace, using the course managers for consultation, and having a course manager review and validate your work after you complete each chapter skill check. There will not be any formal lectures in this course.
You should proceed through the course, chapter by chapter. The chapters should be read in ascending numerical order.

It may be helpful for you review topics in chapters you covered earlier in the course. This can be done quite easily with the navigation buttons.

Skill Checks

At the end of each chapter there will be a skill check, providing you with an opportunity to practice and test your mastery of the skills taught in the chapter.
You must review your skill check solution with a course manager (instructor) before you can go on to the next chapter. When the course manager determines that you have mastered the skills to a satisfactory degree, the course manager will sign you off on that particular chapter, and you will be allowed to proceed to the next chapter.

Chapter Structure

Each chapter in the course covers a particular topic. The first page of each chapter provides an introduction with a brief explanation of what that chapter concerns.
On the left side of the screen you will see several navigation buttons (link buttons) that provide access to various sub-sections within the chapter. You should proceed through the chapter sub-sections in the proper order, by clicking on the navigation buttons from top to bottom. In most cases, that means starting with the chapter objectives.
Towards the top of the screen on most pages in this course you will see two navigation buttons, labeled **Course TOC** and **Chapter TOC**. These buttons provide links to the course table of contents, and the chapter table of contents, respectively.
At the bottom of many of the pages you will find a navigation button that will take you to the top of the page.
Feel free to consult with a course manager at any time as you go through the modules. The course managers there to assist you, and to answer any questions you may have. Most chapters have skill checks at the end to allow you to check your knowledge of the skill covered in the chapter. **Do not complete more than two chapters without getting course manager skill check sign off.**

Work Location

Although this course material is accessable with any web browser on any PC, you are required to do the majority of your course work in the training classroom. If you have a need to do a considerable amount of the course work outside of the classroom, talk to one of the course managers about your situation.
The main reason we recommend that you work in the classroom is to have the course managers more readily available to assist you if necessary.

Challenging a Chapter

As you begin any of the chapters, you may realize that you already possess the skills being taught within the chapter. In such cases, it is permissible for you to go straight to the skill check, complete it and have it signed off by a course manager. This is another advantage of participating in a self-directed course.

Resources

Resources available to you are the course managers, your associates in class, and various reference materials available in class. The chapter on documentation lists reference resources available that you can use to read further about a topic. It is recommended that you take full advantage of these resources during the course, as well as after the course.

CHAPTER 2

Workstation Adjustments

You will need to check a few settings on your classroom workstation to insure that it functions properly when you are writing and testing TPF C code.

There are two areas of concern...

- Having your CMS A-disk set up to work with C
- Having your workstation keyboard mapped properly

To do both of these things, click on the following link and follow the instructions outlined at the Galileo intranet's ISO-C home page...

Remap your keyboard with Extra! 6.x

NOTE: You may find that the classroom keyboard has already been remapped to work with C. If so, then you can skip the instructions concerning the keyboard mapping. However, be sure to follow the instructions concerning updates to your VM profiles.

NOTE: The keyboard back at your office will also need to be remapped. So you'll want to follow these same keyboard remapping instructions when you get back to your office.

CHAPTER 3

Introduction to TPF C

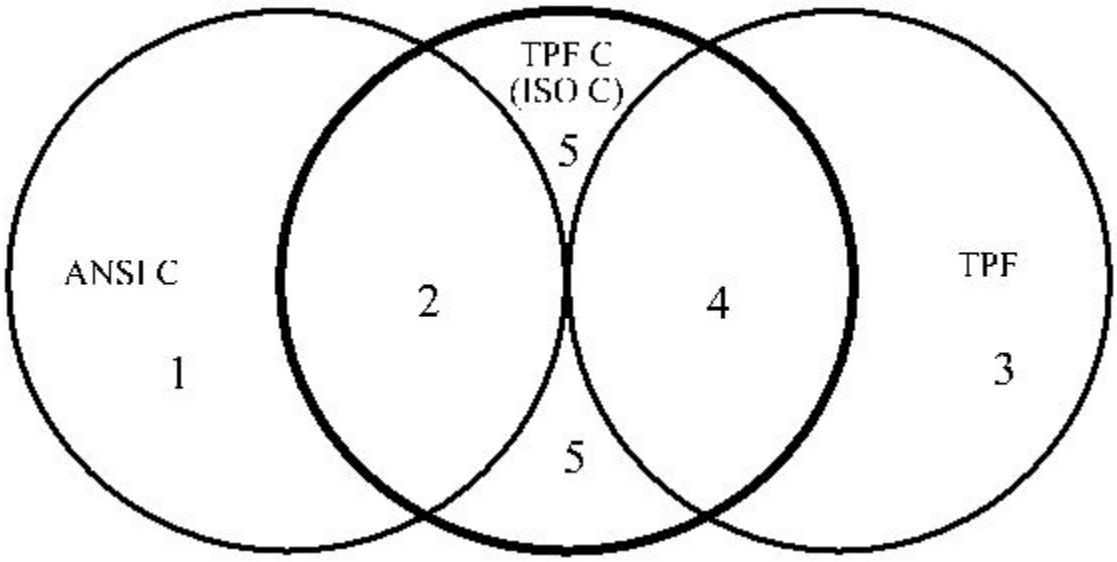
Objectives

The purpose of this chapter is to introduce you to TPF C, as well as to the TPF C guidelines, standards and conventions that have been established at Galileo. After completing this chapter, the student will be:

- Familiar with Galileo's TPF C coding guidelines and standards.
- Familiar with Galileo's TPF C naming conventions.

Merging C and TPF

When learning about TPF C, it might be helpful to view it as being composed of three major pieces. Let's explain what the pieces are, and how they are related.
Most of the students taking this course learned C language on the PC platform. That version of C is often referred to as ANSI C, meaning that it's the version of C that is endorsed by the American National Standards Institute (ANSI). In other words, that version of C complies with the C language standards established by the official ANSI committee.
Rather than being based on ANSI C standards, this TPF C course is based on ISO C standards, which were established by the International Standards Organization (ISO). Most of the ANSI C standards were rolled into the ISO C standards. So the ISO C standards are very much like the ANSI C standards, except for the fact that the ISO C standards apply to C language in the TPF operating system environment, rather than to C language in the Windows or DOS operating system environment found on a PC.
The three major pieces that make up TPF C are the following: the characteristics of ANSI C, the characteristics of ISO C, and the characteristics of what we might think of as traditional TPF. The following picture is intended to represent how those three pieces are interrelated...



In the above picture, you see three circles that overlap each other. From left to right, the circles represent ANSI C, ISO C, and TPF. We've placed the numbers 1, 2, 3, 4, and 5 in various sections of the picture to help explain some specifics. Now let's take a closer look at each circle...

The ANSI C Circle

Within the ANSI C circle we have sections 1 and 2.
Section 2 represents the area where ANSI C and TPF C overlap. In other words, ANSI C has a lot in common with TPF C (often referred to as ISO C). Many of the C standards for the two languages are the same. For example, the syntax for declaring variables is the same. And the way functions are defined is the same.

Section 1 represents that part of ANSI C that is unique to ANSI C. For example, as things stand right now, the graphics related functions that exist in ANSI C are unique to ANSI C, in the sense that TPF C has no such functions. Another example is that all ANSI C programs are required to contain a main() function. This is not true of TPF C.

The TPF Circle

Within the TPF circle we have sections 3 and 4.
Section 4 represents the area where TPF and TPF C overlap. In other words, traditional TPF and TPF C share many common features. When we use TPF C, we're still working with TPF operating system, so the basic TPF architecture prevails in many ways. As with TPF assembler language, we will be accessing ECBs and core blocks with the TPF C language.

Section 3 represents that part of TPF that is unique to TPF. We're referring to the fact that, for most of us, programming in the TPF environment has traditionally meant writing assembler code. This "assembler code" part of TPF is in many ways separate and distinct from ANSI C or TPF C. Examples are the use of BEGIN and FINIS statements, as well as the concept of "enter with no return" (ENTNC). These do not exist in TPF C or ANSI C.

The TPF C (ISO C) Circle

Within the TPF C circle we have sections 2, 4, and 5.
Sections 2 and 4 repesent the fact that TPF C has many features that are common to both ANSI C and traditional TPF. We've given examples of that commonality above.
Section 5 represents that part of TPF C that is unique to TPF C. For example, during this course you will become familiar with functions built into the TPF C language for the purpose of obtaining core blocks, and carrying out file input and output. These functions are unique to TPF C. You won't find them in ANSI C, and even though they are similar to TPF assembler language macros, they are different in many ways. Another way in which TPF C is unique is in terms of its link, load, and test procedures.

All of the similarities and differences alluded to above will become clearer as you progress through this course. You may wish to re-read this section later in the course, when you have a greater understanding of TPF C, and can better relate to the similarities and differences described.

TPF C Coding Guidelines

When writing TPF C programs in this course, and at Galileo in general, you should adhere to the C guidelines and standards set forth in the book ***C Programming Guidelines, Second Edition for Standard C***, by Thomas Plum. Typically, this book is given to students when they attend Galileo Technical Training's "Introduction to C Programming" course. If you don't have the book, a course manager can provide you with advice on how you might obtain it.
For the purpose of indicating Galileo exceptions to the guidelines and standards outlined in the Plum book, as well as for indicating additions and clarifications, an addendum to book has been created. You should read the addendum, which is located at the Galileo TPF C Coding Guidelines and Standards area of the Galileo intranet. (After you go to the guidelines and standards page, you'll have to use the back button on the browser to get back to this page.)

TPF and C Language Terminology

C language includes some terminology similiar to TPF terminology. This can create confusion, especially when the meaning of a term associated with TPF assembler language is quite different than the term's meaning in TPF C language.
The following descriptions are intended to clarify some of the differences and similarities in terminology.

Program, Segment, Module and Function

In TPF's assembler environment, the terms "program" and "segment" are common.

In TPF's C environment, the terms "program," "module," and "function" are common. Generally, a TPF C module is considered to be a unit of code that exists as a separate file. A module contains one or more functions. A TPF C program is made up of one or more modules. The term "segment" is typically not used in TPF C.

Macros and Functions

In general, TPF assembler language related application macros or system macros have been implemented in TPF C as functions. For example, the functionality provided through TPF's GETCC macro is provided in C language as the getcc() function. As in this example, it's quite common for an assembler macro name to be the same as the equivalent C language function name.

Assembler related data macros (DSECTs) have typically been implemented in TPF C as C language structures and unions.

C language does have macros, but they are somewhat different from assembler macros, and are not the same thing as a C language function. These macros are created

through a #define preprocessor directive, which generates in-line code.

DLM

DLM is the acronym for Dynamic Load Module. It is created by linking separately compiled TPF C object modules (somewhat analogous to an assembler loadset). Object modules are compiled from source modules. A source module may contain one or more functions.

When a DLM is created, the link-edit process uses what's called a built script to specify the modules that are to be included in the DLM. The build script is similar to a "make file" commonly used in C compilers for the PC platform.

Parameters and Arguments

TPF assembler related macros often include parameters, and TPF C functions often pass and receive arguments.

The terms "parameter" and "argument" are often considered synonymous. However, in some circles, information passed in a function call is considered to be an argument, while the same information received by the called function is considered to be a parameter.

A further C distinction that is sometimes made is to refer to arguments in a function call as "actual arguments," while arguments specified in the receiving function's function header are referred to as "formal arguments."

DSECTs vs. Structures and Unions

Structures and unions are the TPF C equivalent of TPF data macros (DSECTs).

Globals

There are two meanings to the word "global" in TPF C. The term is used to describe the scope of a variable declared outside a function, indicating that the variable has a broad scope and is known "globally." The other meaning refers to TPF globals (specific main storage data fields and records), which can be accessed through various TPF C global functions.

Function Types

Entry Point Function

The function positioned as the first function in the DLM (Dynamic Load Module) is the DLM's entry point function. Such a function can be called by other functions in the same DLM or in another DLM. This is the only

point of entry into a DLM, and the name of the DLM must be the same as the name of the entry point function.

External Function

External is the default storage class for functions. An external function can be called by any other function within the same DLM. In addition, if an external function happens to be the entry point function, it can be called from outside the DLM.

Static Function

Functions of the static storage class have a more limited scope than external functions. They are known only within the source module in which they are defined. Such functions are declared and defined with the "static" keyword.

Library Function

Standard C functions applicable to TPF are available for use.

Examples: scanf(), strcmp()

TPF API functions are provided by IBM. These are C equivalents of TPF assembler related macros.

Examples: getcc(), flipc()

Installation coded macros may also be deployed as library functions.

Examples: date and ordinal number conversion routines.

Naming Conventions

The following describes some of the naming conventions that have been established at Galileo International:

DLM (Dynamic Load Module)

The name of the DLM (the primary four character name) must match the name of the entry point function (the first function in the DLM).

Header Files

TPF C header files should have an "h" file type.

Example: myhead h

Preprocessor Directives

Leading or trailing underscores in preprocessor directives are reserved for systems use, and should not be coded by application programmers.

Example:

```
#ifndef __eb0eb__

#define __eb0eb__
```

Source Code Files

TPF C source code files should have a "c" file type.

Example: pcx5t2 c

Structures Mapped to DSECTs

The letter "c" should be added as a suffix to the name, right before the version designation.

Example: wa0aact2

Functions within a C Program

Use a four character TPF-like function name.

Example: pcx5()

or...

Use a mapped C function name.

Example: edit_address()

Variables

Leading or trailing underscores in the names of variables and structure or union members are reserved for systems use, and should not be coded by application programmers.

Examples:

```
#define ce1fa2
_farw2._long2._ce1fa2

#define ebcfa2
_farw2._long2._ebcfa2
```

Variable names starting with "c" are reserved for IBM use.

Additional naming conventions for variables and functions are described at the ISO-C home page on the Galileo intranet.
Additional naming conventions for header files are described there, as well.

CHAPTER 5

Editing, Compiling and Testing C Code in the Galileo International TPF Environment

Objectives

- .

Editing a C source module

Once we are in XEDIT, we can input the code.

When we are done inputing code, we will file the new source module to the A disk.

Compiling a C source module

OS/390 Compiler Options

The MVS Compiler that Galileo International uses allows certain features to be changed via an input file (ASMOPTS file). These options are normally controlled by the PROASM Environments, but you can change them by creating an ASMOPTS file on your A-Disk. THIS IS POSSIBLE BUT NOT RECOMMENDED.
To create this file, you will use XEDIT. The file name must match the PROASM Environment. The file type must be ASMOPTS. Once you have created the file, you will use Keywords and parameters to tell the compiler which options you wish to use. The exact layout of the file (and additional information) can be obtained from the LAN version of BookManager (use Galileo Workstation to install or execute the product).

Look for:
Bookshelf - *C/C++ Optional Feature Bookshelf*
Book - *C/C++ User's Guide*
Page - *2.3 Chapter 6 Compiler Options*

Beware...some options are tied to other options and changing one option may automatically change another option. Read all of section 2.3 before attempting to change compiler options!!!!
AGAIN -- Compiler options are changeable, but it is recommended that you do not change them.

Building the BSCR file for your DLM

Summary of Steps from Creating a Source Module to Testing a DLM

Creating a Source Module

1. XEDIT the source module, following GI naming convention standards.
   example: **X PGM1T1 C**
2. Save the changes after completing your code. From the XEDIT command line, type **File**.

Compiling a Source Module

3. Use PROASM to compile. You may either use the ISOCMVS environment, TAPCMP environment or create a private environment.
   example: **PROASM TAPCMP PGM1T1**

   Don't forget to check the Message Summary for errors!
4. Create your SourceView Control File. Go into the reader list and use the **SVIEW** command next to your returned Source Listing.

   You need to do this each time you update your Source Code!

Creating a DLM

5. Build a BSCR file. Use the name of the first Source Module as the DLM name, adding "BS" to create a proper DLM name. Use XEDIT to add the Source Module and Stublist components.
   example: **X PGM1BST1 BSCR**

   You should have 2 sections to your BSCR, the "*DLM dlmname*" and the "*STUBLIST*".
6. Save the changes after completing the BSCR file. From the XEDIT command line, type **File**.
7. Now create the DLM by using the CBLD Link/Editor.
   example: **CBLD PGM1T1**

   Don't forget to check the JCL return file condition codes!

Testing a DLM

8. Use TASTE to do your intial testing.
9. See a Course Manager for further instructions on using TASTE to test your DLM.

TPF and C Language Terminology

C language includes some terminology similiar to TPF terminology. This can create confusion, especially when the meaning of a term associated with TPF assembler language is quite different than the term's meaning in TPF C language.
The following descriptions are intended to clarify some of the differences and similarities in terminology.

Program, Segment, Module and Function

In TPF's assembler environment, the terms "program" and "segment" are common.

In TPF's C environment, the terms "program," "module," and "function" are common. Generally, a TPF C module is considered to be a unit of code that exists as a separate file. A module contains one or more functions. A TPF C program is made up of one or more modules. The term "segment" is typically not used in TPF C.

Macros and Functions

In general, TPF assembler language related application macros or system macros have been implemented in TPF C as functions. For example, the functionality provided through TPF's GETCC macro is provided in C language as the getcc() function. As in this example, it's quite common for an assembler macro name to be the same as the equivalent C language function name.

Assembler related data macros (DSECTs) have typically been implemented in TPF C as C language structures and unions.

C language does have macros, but they are somewhat different from assembler macros, and are not the same thing as a C language function. These macros are created through a #define preprocessor directive, which generates in-line code.

DLM

DLM is the acronym for Dynamic Load Module. It is created by linking separately compiled TPF C object modules (somewhat analogous to an assembler loadset). Object modules are compiled from source modules. A source module may contain one or more functions.

When a DLM is created, the link-edit process uses what's called a built script to specify the modules that are to be included in the DLM. The build script is similar to a "make file" commonly used in C compilers for the PC platform.

Parameters and Arguments

TPF assembler related macros often include parameters, and TPF C functions often pass and receive arguments.

The terms "parameter" and "argument" are often considered synonymous. However, in some circles, information passed in a function call is considered to be an argument, while the same information received by the called function is considered to be a parameter.

A further C distinction that is sometimes made is to refer to arguments in a function call as "actual arguments," while arguments specified in the receiving function's function header are referred to as "formal arguments."

DSECTs vs. Structures and Unions

Structures and unions are the TPF C equivalent of TPF data macros (DSECTs).

Globals

There are two meanings to the word "global" in TPF C. The term is used to describe the scope of a variable declared outside a function, indicating that the variable has a broad scope and is known "globally." The other meaning refers to TPF globals (specific main storage data fields and records), which can be accessed through various TPF C global functions.

Function Types

Entry Point Function

The function positioned as the first function in the DLM (Dynamic Load Module) is the DLM's entry point function. Such a function can be called by other functions in the same DLM or in another DLM. This is the only point of entry into a DLM, and the name of the DLM must be the same as the name of the entry point function.

External Function

External is the default storage class for functions. An external function can be called by any other function within the same DLM. In addition, if an external function happens to be the entry point function, it can be called from outside the DLM.

Static Function

Functions of the static storage class have a more limited scope than external functions. They are known only within the source module in which they are defined. Such functions are declared and defined with the "static" keyword.

Library Function

Standard C functions applicable to TPF are available for use.

Examples: scanf(), strcmp()

TPF API functions are provided by IBM. These are C equivalents of TPF assembler related macros.

Examples: getcc(), flipc()

Installation coded macros may also be deployed as library functions.

Examples: date and ordinal number conversion routines.

Chapter 6

Introduction to TPF C Listings

Objectives

The purpose of this chapter is to introduce you to the TPF C Source and DLM Listings. After completing this chapter, the student will be able to:

- Determine some of the options used to compile the source module.
- Locate various sections of the source module listing.
- Use the cross reference listing to locate an identifier in the source code.
- Determine the offset of a structure member.
- Locate various sections of the DLM listing.

Source Listing Components

Getting Started

Before you start this section, contact a course manager to obtain a copy of the Source Listing referenced in this section.

You do not have to print a listing.

Overview of the major sections of a listing

***** P R O L O G *****
***** S O U R C E *****
***** I N C L U D E S *****
***** C R O S S R E F E R E N C E L I S T I N G *****
***** S T R U C T U R E M A P S *****
***** M E S S A G E S U M M A R Y *****
***** P S E U D O A S S E M B L Y L I S T I N G *****
***** S T O R A G E O F F S E T L I S T I N G *****

The PROLOG Section of the C Source Listing

The Prolog contains information about how and when the source module was compiled. If you reference page 1 of the hard copy of the listing, you will see titles on the left side of the page. The titles include **Compile Time Library**, **Command Options**.
The area of interest to us now is the **Compiler Options** area. It tells us what options were used to create the listing.
Some of the compiler options are described in the table below...

Option	Purpose	Comments
TARGET (TPF) () (LE) (COMPAT)	Target Environment	Generates TPF prefix, prolog and epilog code and program linkages in the object module

SOURCE	Source Listing	Generate listings with source code statements.
LIST	Pseudo Assembler Listing	Include a pseudo assembler listing as part of the output.
FLAG (I)	Minimum Error Message Severity Level	Messages are immediately following the error source line and at the end of the listing I - Informational W - Warning E - Error S - Severe Error
OPTIMIZE(0)	Optimization Level	Optimize for faster execution and less main storage. 0 - No Optimization 1 - Minimal Optimization 2 - Load to Production Optimization
AGGR	List Aggregate Data Types	Include all maps of structures and unions in the listing.
SHOWINC	Show #Include Files in the Listing	Generate a listing of all #include files processed, in the compiler and pseudo assembler sections.
EXPMAC	Show expanded Macros	Show the expansion of all macros (#defines) in the source code.
XREF	Cross Reference Listing	Include a cross reference table of names used and line numbers where they are declared or referenced
SEARCH(...)	Search sequence-system #include "filename"	Standard Search
LSEARCH(...)	Search sequence-user #include "filename"	Standard Search
RENT	Re-entrant code generation	Compiler makes code Re-entrant, if it is not already.

These are just a few of the compiler options. For more information about compiler options, see the following page in the LAN version of BookManager...

Bookshelf - *C/C++ Optional Feature Bookshelf*
Book - *C/C++ User's Guide*
Page - *2.3 Chapter 6 Compiler Options*

Our compiler options of note in the example listing are **XREF**, **AGGR**, **EXPMAC**, **SHOWINC** and **OPTIMIZE(0)**.

The SOURCE Code Section of the Listing

In our example listing, the SOURCE section is on page 2 and continues to page 8.

This section is the actual source code with any error or warning statments.

The headers for this section are **LINE**, **STMT**, a scale (***...+....1...**), **SEQNBR** and **INCNO**.

LINE is the sequence number, within each file. So, if you use the SHOWINC compiler option (expanding the #include files), the Line number will count the number of lines in your source file and the number of lines included with each header file.

For the example below, you hae 8 lines in the source module file before the first #include. The 9th line is then LINE 1 of the giFIRST.h file. The count continues until the end of the giFIRST.h file (line 181). The next LINE is the 9th line of the source module file. The 9th line is another #include, so the next line is LINE 1 of the c$pd1xx.h file.

```
LINE STMT
            *...+....1....+....2....+..
   1       |/*(c)Copyright Galileo Inte
   2       |/*Partnership 1995. All ri
   3       |
   4       |/*pcx1tp, chapter 7, progra
   5       |/*the purpose of this simpl
   6       |/*to show the anatomy of a
   7       |
   8       |#include "giFIRST.h"
   1       |/*
   2       |/* This is the giFIRST.h Ga
   3       |/* included first in every
...
180       |#endif
181       |
   9       |#include "c$pd1xx.h"
   1       |/*(c)Copyright Galileo Interna
   2       |/*Partnership 1995. All right
   3       |
```

giFirst.h starts on page 2 and ends on page 5. c$pd1xx.h starts and ends on page 5. giLAST.h starts on page 5 and ends on page 7.

STMT is the count of actual C Statements. It marks the executable code.

the scale (***...+....1...**) allows you to see the layout of the source code.

SEQNBR is the overall source line number. It will start at 1 and go until the end of the source code. Each line will get a line number.

Our example starts with SEQNBR 1 and goes to 325.

INCNO is the include number. This number tells us which file caused the line to be included. Lines included because they are in the source module file don't have an INCNO. Lines that are included because of the first #include file will be noted with a INCNO of 1. The second #include file lines will be noted with a INCNO of 2, and so on.

In our example, Lines included because of giFIRST.h are noted with an INCNO of 1. Lines included because of c$pd1xx.h are noted with an INCNO of 2 and lines included because of giLAST.h are noted with an INCNO of 3.

The INCLUDES Section of the C Source Listing

This page lists the file(s) that were included when this file was compiled.

In our example, page 9 shows only 3 files were included; giFIRST.h, c$pd1xx.h and giLAST.h.

```
                    * * * * * I N C L U D E S * * * * *

INCLUDE FILES   ---   FILE#    NAME

                        1     GIFIRST H A1
                        2     C$PD1XX H A1
                        3     GILAST H A1

                     * * * * * E N D   O F   I N C L U D E
```

The CROSS REFERENCE LISTING Section of the C Source Listing

This section lists all defined tag names.

There are three headers here; **IDENTIFIER**, **DEFINITION** and **ATTRIBUTES**

IDENTIFIER gives the tag name.

DEFINITION gives the location of the tag definition. The first number is the SEQNBR. The number after the '-' is the INCNO and the number after the ':' is the LINE within the INCNO section.

ATTRIBUTES gives you information (storage class, length and data type) about the tag and cross references to where the tag is defined and referenced. So the first number is where the tag is defined. The following values are locations where the tag is referenced. These references follow the same format at the DEFINITION column.

```
IDENTIFIER          DEFINITION   ATTRIBUTES
                                 <SEQNBR>-<FILE NO>:<FILE LI

exit                 324-0:25      Class = external reference,
                                 Type = C function returning
                                 324-0:25
```

```
local_int           316-0:17     Class = automatic, Length =
                                 Type = signed short integer
                                 319-0:20, 321-0:22, 322-0:2

pcx1                 312-0:13     Class = external definition,
                                 Type = C function returning

pd1xx                197-2:7      Class = tag, Type = structu
315-0:16

pd1xx_local         315-0:16    Class = automatic, Length =
                                 Type = structure pd1xx

pd1xx_local.pd1x2   200-2:10    Class = member, Length = 2,
```

The STRUCTURE MAPS Section of the C Source Listing

A packed and unpacked layout of each item is printed.
The packed layout is exactly the size coded (no pad bytes).
The unpacked layout is how the computer will assign memory (pad bytes). The pad bytes are noted as ***PADDING***.
Our example shows the Structure Maps on page 11.

```
=========================================================
| Aggregate map for: structure pd1xx
|=======================================================
|      Offset        |       Length        | Member Name
|    Bytes(Bits)     |    Bytes(Bits)      |
|====================|=====================|===============
|       0            |        1            | pd1x1
|       1            |        1            | ***PADDING***
|       2            |        2            | pd1x2
|       4            |        1            | pd1x3
|       5            |        1            | ***PADDING***
=========================================================

=========================================================
| Aggregate map for: _Packed structure pd1xx
|=======================================================
|      Offset        |       Length        | Member Name
|    Bytes(Bits)     |    Bytes(Bits)      |
|====================|=====================|===============
|       0            |        1            | pd1x1
|       1            |        2            | pd1x2
|       3            |        1            | pd1x3
```

The NOAGGR compiler option will suppress the display of aggregate data types.

The MESSAGE SUMMARY Section of the C Source Listing

Messages result from pre-processor or compiler errors or warnings. The message appears in the SOURCE section of the listing, immediately after the line that caused the message.

The Message Summary section details how many occurrances of each type of message were found.
To find this section, use the XEDIT search command ('/') to search for "error(". The command to enter on the command line is "/error(". When you hit the enter key, you will see the Message Summary header line on your current line.
In the header line, the first column is **Total** messages, followed by **Informational (00)**, **Warning (10)**, **Error (30)** and **Severe Error (40)**. You want to see zero under each column. The only exception is Informational. Informational messages are warnings of potential errors. You can still create executable code with Informational messages.
In our example, the MESSAGE SUMMARY is on page 12 and there are not messages in the source.

If there were messages, the MESSAGE SUMMARY section would be followed by a MESSAGES section. The MESSAGE section is a list of the messages found in the source. The format is as follows:

```
                                    * * * * * M E S S A
MSG ID      SEV    TEXT
                   <SEQNBR>-<FILE NO>:<FILE LINE NO>
CBC3304     10     No function prototype given for ' '.
                   ( 1416-0:26 )
                            * * * * * E N D   O F   M
```

The **MSG ID** is the compiler message number. The **SEV** is the severity of the message, in this case 10 is a Warning message. The **TEXT** is what is printed in the Source section. The numbers in the parenthesis is the SEQNBR, INCNO and LINE numbers. The format is the same as the Definition column in the Cross Reference Listing section.

The PSEUDO ASSEMBLY LISTING Section of the C Source Listing

This section is the source code followed by the Assembly code that was generated by the compiler.
There are two areas in this section that you need to watch for. The **PROLOG** and the **EPILOG**. Each function will get a PROLOG and an EPILOG.
The main thing the PROLOG does is to tell TPF to reserve enough heap storage for the function's variables. The PROLOG also saves R14 through R4. The PROLOG is executed every time the function is called.
The EPILOG releases the heap storage and restores R14 through R4. The EPILOG also stores the return value in R15, if a return value is used. The EPILOG is executed by the return or exit commands.
Looking at our example, you will not see executable code until the bottom of page 18 and the top of page 19.
Notice that all of the #defines, #includes and other pre-processor directives are listed, but no Assembly code is shown. That's because pre-processor directives do not generate Assembly code.
Each line of Assembly code has an **OFFSET** (displacment from R8) and **OBJECT CODE**. This looks very similar to our Assembly Listings. Between the Assembly lines of code, you will see **LINE#** of code. This is the C source code. The LINE# is the SEQNBR from the Source section.
The OPTIMIZE() compiler option controls the type of Assembly code that is generated. Normally we test with OPTIMIZE(0) (no optimization). This code is very similar to what Assembly programmers would write. OPTIMIZE(2) is what is normally loaded. This Assembly code will resemble systems code. It is much more effecient during execution.

Near the bottom of the Pseudo Assembly listing you will find a summary of the Assembler command used. This is listed under the heading of **IBM / 370 INSTRUCTION USAGE**.
One last area under the Pseudo Assembly Listing is the **EXTERNAL SYMBOL DICTIONARY**. This is a list of external references that will be resolved at Link/Edit time.

The STORAGE OFFSET LISTING Section of the C Source Listing

This section lists all defined variables.
This section is very useful during dump analysis. It is used to help find the values of variables in the dump.
There are three headers here; **IDENTIFIER**, **DEFINITION** and **ATTRIBUTES**
IDENTIFIER gives the variable name.
DEFINITION gives the location of the definition. The first number is the SEQNBR. The number after the '-' is the INCNO and the number after the ':' is the LINE within the INCNO section.
ATTRIBUTES gives you the storage **Class**, **Offset** and **Length** of the variable.

> **Class =** is the storage class.
> **Offset =** is the offset (displacement) from the beginning of the Heap Storage for this Function.
> **Length =** is the length of the variable.

```
IDENTIFIER          DEFINITION   ATTRIBUTES
                                 <SEQNBR>-<FILE NO>:<FILE LI
local_int           316-0:17      Class = automatic,

pd1xx_local         315-0:16     Class = automatic,

pd1xx_local.pd1x2 200-2:10     Class = automatic,
```

DLM Listing Components

Getting Started

If you wish to see the example used below, from the CMS Ready Prompt enter "**LISTING LDRH**". When the LISTING Screen comes up, move your cursor to the file "**LDRHBS6A.APO**". Use **PF11** to browse the listing.

Using a DLM Listing

The DLM Listing is most useful for Dump Analysis.
You will use the DLM Listing to determine:

> What source modules are included in the DLM.
> How static memory is layed out.
> Where each function starts and ends in the program block.

Overview of the major sections of a listing

```
=========================
|                Prelinker Map
|
=========================
*** M O D U L E   M A P ***
C R O S S - R E F E R E N C E   T A B L E
```

Prelinker Map

This area is sub divided into three areas; File Map, Writable Static Map and ESD Map of Defined and Long Names.

File Map

This section lists the Source Modules of the DLM. You will look at the FILE NAME column to determine the Source Module Names.

In our example, FILE ID 00001 is CSTRTD40. This is a file that TPF inserts to create the start of a DLM.

In our example, we need to look at FILE ID 00003 to see the first Source Module, LDRH53. From here we see LDRIG7 and LDRJG8 are the other two Source Modules. ZZVM51 may or may not be a viable Source Module. To determine ZZVM51, we would need to look at the Source Listing.

The remaining list are items that TPF uses and are of little value to most application programmers.

```
================================================
|                                  File Map
================================================

*ORIGIN  FILE ID  FILE NAME

   PI     00001   DD:OBJLIB(CSTRTD40)
   P      00002   DD:SYSIN
   PI     00003   DD:OBJLIB(LDRH53)
   PI     00004   DD:OBJLIB(LDRIG7)
   PI     00005   DD:OBJLIB(LDRJG8)
   PI     00006   DD:OBJLIB(ZZVM51)
   A      00007   PD.TPF41.STUB(CEESG003)
   A      00008   PD.TPF41.STUB(CEESTART)
   A      00009   PD.TPF41.CLIB(EXIT)
...
```

Writable Static Map

This section lists gives us the information needed to determine the beginning of the Static area for each of the Source Modules.

The FILE ID column in this section corresponds directly to the FILE ID column in the File Map section. So, from our example FILE ID 00003 could be cross referenced with the File Map section to find the name is LDRH53.

The OFFSET column tells us how far (displacement) into the Writable Static Area (WSP) we must look to find the beginning of the Source Module's static memory. This is a hexadecimal value.

The LENGTH column tells us the length of the Source Module's static memory. This is a hexadecimal value.

```
================================================
|                          Writable Static Map
================================================

OFFSET    LENGTH   FILE ID   INPUT NAME

     0        20    00003     @STATIC
    20       120    00004     @STATIC
   140       1C8    00005     @STATIC
```

ESD Map of Defined and Long Names

This section is of little value to most applications programmers.

*** M O D U L E M A P ***

This area is useful during dump analysis to determine how the program block can be divided.
The SECTION OFFSET column is not of much value to most application programmers.
The CLASS OFFSET column can be used to determine where each user defined function starts. This is a hexadecimal value.
The NAME column is the name of the function.
The TYPE column is not of much value to most application programmers.
The LENGTH column is the length of the function.
The remaining columns are not of much value to most application programmers.

```
*** M O D U L E   M A P ***

---------------
CLASS B_TEXT            LENGTH =       2670  ATTRIBUTES =
---------------

SECTION    CLASS
  OFFSET   OFFSET  NAME                 TYPE      LENGTH  D
```

```
            0   @@DLMHDR          CSECT          50   S

           50   @@LM0001          CSECT           8   S

           58   $PRIV000010       CSECT         1A8   S
   40      98  LDRH                  LABEL
```

```
          200   @@LM0002          CSECT           8   S

          208   LDRIG7            CSECT         938   S
    0     208  LDRI                  LABEL
  750     958  CHECK@CR              LABEL
```

CROSS-REFERENCE TABLE

This section used to be valuable for determining V-CON displacements, which was then used to locate displacements into Stack and Static memory. These calculations are no longer necessary, because of updates to the dump format and the addition of the Module Map in the DLM Listing.
So, this section is not of much value to most application programmers.

CHAPTER 7

TPF C Preprocessor Directives

Objectives

The purpose of this chapter is to familiarize you with various preprocessor directives that you will use or at least encounter in TPF C. After completing this chapter, the student will be able to use preprocessor directives to do the following:

- Insert header file information into source files.
- Define a symbolic constant using a #define directive.
- Conditionally pass code to the compiler by using preprocessor directives.

Preprocessor Directives

Preprocessor directives are commands that are received and interpreted by the preprocessor portion of the compiler, during the compiling of a program. Preprocessing takes place before compilation.
The preprocessor looks for preprocessor directives in the source code of the program being compiled. These directives tell the preprocessor to carry out certain actions, such as...

- Replace a symbol in the current file with a specified replacement value
- Imbed a file within the current file
- Conditionally compile a section of the current file
- Generate a diagnostic message

For example, the #include directive tells the compiler to insert another file into the current file. (In other words, into the file containing the #include directive).

All preprocessor directives start with the # symbol (pound sign or hash symbol).

As you go through this chapter, you will learn about many of the preprocessor directives that exist.

The Syntax for Preprocessor Directives

The following syntax rules must be followed when you code preprocessor directives in a TPF C program or header file:

- Preprocessor directives must begin with the pound sign or hash symbol (#), otherwise they will not be recognized as preprocessor directives.
- The pound sign or hash symbol must be the first non-blank character on a line. In other words, nothing can come before the pound sign on that same line, other than blank spaces.
- No other statement can be on the same line with a preprocessor directive, except for a comment, which must be to the right of the directive.
- Unlike most other C language statements, preprocessor directives do not end with a semi colon.

- A backslash character (\) is used to indicate that a preprocessor directive continues on to the next line. This continuation indicator is used when the text of a directive is too long to fit on one line.
- A preprocessor directive can affect only statements that come after it. In other words, the scope of affect is forward from the location of the directive. A directive generally is not applied to statements that come before it.

The #include Preprocessor Directive

The **#include** preprocessor directive is used to specify header files that are to be included with the source code program for compilation.
The format of the directive is as follows:

```
#include <header file name>

or

#include "header file name"
```

The Galileo C Coding Standards, which were written with mainframe compiles in mind, state that Galileo headers should enclosed in double quotes and IBM headers in angled brackets. This roughly follows the user versus system distinction. The names of header files should be in lower case.

Go to the #include syntax section of the ISO-C home page for additional important information.

The gifirst.h and gitpf.h Header Files

gifirst.h and **gitpf..h** are two Galileo header files that must be included in every TPF C program. The **gifirst.h** file must be your first header file.

The Order of Header Files

It is generally accepted practice to sequence include files starting at system level and finishing at application-specific headers to aid readability. The suggested order of include files for the TPF environment is shown in the following code example...

```
#include "gifirst.h"    /* Galileo first header file */
                        /* for PC compile compatibility */

#include <stdio.h>      /* IBM-provided ANSI C header files */
#include <stdlib.h>
#include <string.h>

#include <tpfeq.h>      /* IBM-provided TPF specific header files */
#include <tpfapi.h>
```

```
#include <tpfio.h>

#include "gitpf.h"      /* Galileo system level definitions */
                        /* like TRUE, FALSE _*/

#include "fmsgc.h"      /* Application unique header files */
#include "wa0aac.h "
```

The #define Preprocessor Directive

There are four primary purposes for which a #define directive is used:

- To create a symbolic constant
- To create a function-like macro
- In combination with an #undef directive
- To create a more user-friendly function name

To Create a Symbolic Constant

In its simplest form, the #define preprocessor directive works much like a search and replace operation in a word processor. In the #define statement, a symbolic constant is defined as being equivalent to a certain value. The programmer then codes the symbolic constant in appropriate places in the program. As a result, when the program is compiled, the preprocessor finds every occurrence of the symbolic constant, and replaces each occurrence with the constant's equivalent value.
For example, let's say we're writing a program that will make use of the mathematical constant commonly referred to as pi. We wish to create a symbolic constant "PI" and to equate it with the value 3.14. To do so, we would code the following line in our program or header file:

```
#define PI     3.14
```

The above preprocessor directive will allow us to use the symbolic constant PI in our code, rather than directly using the value 3.14. For example, we might code the following assignment statement in our program, at some location after the spot where the PI symbolic constant is defined:

```
result = 300 * PI;
```

By using the symbolic constant PI, our code becomes more readable than it would be if we used the constant 3.14, in the sense that PI is more meaningful than 3.14.
Using the symbolic constant also makes maintenance easier, as it allows us to globally change the value of pi in our program, simply by changing the value to which the symbol PI is equated in our #define statement. For example, let's say it becomes necessary to change the value of pi to 3.1415. To make the change we just change the #define statement to the following, and re-compile the program:

```
#define PI     3.1415
```

As a result, for every statement in the program that uses the constant PI, the value 3.1415 will be substituted at compile time, rather than 3.14. We implemented the change throughout the entire program by changing only one statement, that being the #define statement. If, instead, 3.14 had been hard coded in the program, we might have had to make dozens of changes (from 3.14 to 3.1415) in the program - - a change at every location where we had hard coded the value 3.14.
Using uppercase letters for the sybolic constant name (in this case, PI) is a common convention that you should follow, the purpose of which is to make symbolic constants more easily recognizable.
Another thing to remember is that the scope of influence for a symbolic constant is from the location of the #define directive on downward through the file.

To Create a Function-Like Macro

In C, a macro (sometimes referred to as a function-like macro) carries out particular processing, much like a function. By using a #define directive, the macro name is defined and equated to an expression that represents the processing or functionality to be carried out by the macro. When the macro name is recognized in the source code, it's treated as a macro call, and that particular processing is invoked.
The difference between a macro call and a function call is significant. A function call passes an argument ***value*** to the function while the program is running, whereas a macro call passes an argument ***token*** to the program before compilation. The two processes are different, and they take place at different times.
Here is the general format used to define a macro:

#define *macro_name(parameter list) expression*

As an example, let's say we want to create a macro called **`SQUARE`** that accepts an input value, then multiplies that value by itself, and then returns the resulting value to the point of call. The following is an example of such a macro definition:

```
#define SQUARE(value)  value * value
```

The **`#define`** is required, and the **`SQUARE(value)`** that follows it indicates the name of the macro, as well as its parameter list. Notice that the name of the macro is in uppercase letters. The name of a C macro is in uppercase by convention. Also notice that the left parenthesis immediately follows the macro name with no intervening whitespace.
The parameter list indicates how many parameters will be provided when the macro is called. In our example, one parameter is specified, and that parameter is represented by the name **`value`**. You can also have more than one parameter, each separated by a comma, or you can have none, in which case the area between the parentheses is empty. A parameter name in the macro definition can be any name, as long as it's a legal C language variable name.
Following the macro name and parameter list is the expression that the macro name and parameter list equate to. In our example, the expression is **`value * value`**. The expression may include parentheses to force precedence or to make things clearer. The expression typically includes the parameter names specified in the

parameter list. However, there is no requirement that the expression include those names.
Now let's expand our example to include both a macro definition and a macro call, as shown below:

```
#define SQUARE(value)  value * value  /* definition */
...
...
short count = 2;
...
...
printf("%hd squared equals %hd",count, SQUARE(count));
...
...
```

The macro call is the **SQUARE(count)** that is specified as an argument in the printf() call. The parameter in the macro parameter list is the variable **count**, and it is considered to be synonymous with **value** specified in the macro definition.
The preprocessor recognizes the macro call in the printf() call, because of the **#define** statement that defines the macro. The preprocessor then replaces the macro call with the corresponding expression specified in the macro definition. When the expression is substituted for the call, the parameter in the call (in this case, the variable **count**) is substituted into the expression wherever the word **value** appears. In affect, the **SQUARE(count)** macro call is replaced with the following expression:

```
count * count
```

In our example, once that substitution is made by the preprocessor, then the printf() call, in effect, looks like the following:

```
printf("%hd squared equals %hd",count, count * count);
```

When the arguments in the printf() call are evaluated, the variable **count** contains the value 2, so the value 2 is substituted for **count**, and we have the following output as a result of the printf() call:

```
2 squared equals 4
```

The **SQUARE** macro could be called from anywhere within our example program, with a short integer variable or short integer value provided as the parameter in the call. In general, whether a parameter specified in a macro call must be a variable or a value depends upon the nature of the expression specified in the macro definition.
Macros can have more than one parameter, as the example below shows:

```
#define TOTAL(number1,number2) number1 + number 2
short num1 = 3, num2 = 6;
short num3;
num3 = TOTAL(num1,num2);        /* macro call */
printf("%hd",num3);              /* num3 equals 9 */
...
...
```

Macros are often faster and more efficient than functions, because macros result in 'inline' code, and do not generate the overhead that function calls generate. However, macros can cause significant increases in program size. That's because for each macro call, the preprocessor will insert into the program the code associated with the macro. So if you call a macro 10 times, then 10 lines of code will be inserted into the program. In contrast, if you call a function 10 times, you'll have just one copy of the function statements in your program.

In Combination with a #undef Directive

A **#define** directive is sometimes followed later in the source code by an **#undef** directive, so that the previously defined symbolic constant is no longer active. In other words, the #undef allows you to undo the previous defining of a symbolic constant. For an example, see the "#undef" section of this chapter.

To Create a more User-Friendly Function Name

You can use a **#define** directive to give a more user-friendly alias to an external reference, such as in a function call. For example...

```
#define CONVERT_DATE UCDR

...
...

CONVERT_DATE();        /* really calling UCDR */
```

In this example, when we call **CONVERT_DATE()** we are really calling the TPF assembler program UCDR. But the symbolic name may be a more meaningful name.

The #undef Preprocessor Directive

As its name suggests, **#undef** undefines a name that was defined with a previous #define.
If, for example, the following directive is provided in the source code...

```
#undef PI
```

then from that point on, **PI** does not exist as a symbolic constant.
The purpose of the #undef directive is to undefine or "turn off" a symbolic constant that was previously defined with a previously stated **#define** preprocessor directive.

Preprocessor Conditional Directives

Preprocessor conditional directives provide a way for us to implement logic to effect the behavior of the preprocessor. These directives are usually applied to control two major areas of activity:

- Determining whether a preprocessor directive is to be processed by the preprocessor, or whether it is to be ignored
- Determining whether the preprocessor is to pass on to the compiler particular source code statements

The conditional directives are read by the preprocessor, and the logic is implemented through "if - else" types of constructs...

Conditional Directive	**Purpose**
#if	Invokes the logic of a typical "if" statement.
#ifdef	Invokes "if defined" logic
#if defined	Same as above
#ifndef	Invokes "if not defined" logic.
#if !defined	Same as above
#elif	Invokes the logic of a typical "else if" statement
#else	Invokes the logic of a typical "else" statement
#endif	If you use any of the above conditional directives, **you must close the construct with this directive**.

Click on each of the following links to see important examples of how the conditional directives can be used...

- Example1
- Example2

Example 1

This example shows how conditional directives are often used to control which header files are included with a program's source code.
Below you see the partial contents of two header files, **header1.h** and **header2.h**. The first header file contains a preprocessor directive that defines "GALILEO". The second header file contains a preprocessor directive that defines "APOLLO".

```
/* Partial contents of header1.h */

. . .
. . .

#define GALILEO
```

```
. . .
. . .
```

/* Partial contents of header2.h */

```
. . .
. . .

#define APOLLO

. . .
. . .
```

Below is a third header file (**header3.h**) containing preprocessor conditional directives which raise the question of whether "GALILEO" and "APOLLO" have previously been defined.

/* Partial contents of header3.h */

```
. . .
. . .

#ifdef GALILEO

     #include <galileo.h>

#ifdef APOLLO

     #include <apollo.h>

#endif        /* don't forget the required #endif */

. . .
. . .
```

The three header files above can be included in various ways in a source code program such that the conditional directives cause **galileo.h** to be included in the source code. The partial source code in the box below is such an example...

/* Partial contents of pcx1t3.c */

```
. . .
. . .

#include <header1.h>            /* defines GALILEO */

#include <header3.h>

. . .
. . .
```

The above **pcx1t3.c** program will end up with **galileo.h** included rather than **apollo.h** included. That's due to the fact that **header1.h** was included in **pcx1t3.c**, and therefore **GALILEO** was indeed defined. The file **header3.h** provides the logic that includes **galileo.h**, provided that **GALILEO** was defined.

In contrast, if we wanted the conditional directives to cause **apollo.h** to be included in the source code, then we would include **header2.h** in the source code file, rather than **header1.h**, so that **APOLLO** will be defined rather than **GALILEO**. The partial source code in the box below is such an example...

```
/* Partial contents of pcx1t3.c */

. . .
. . .

#include <header2.h>            /* defines APOLLO */

#include <header3.h>

. . .
. . .
```

Our Example 1 shows how preprocessor conditional directives can be used to implement logic to control whether certain directives are processed by the preprocessor. More specifically, our example demonstrated how we might control which of two header files is included in a source code program at compile time.

Example 2

This example shows how conditional directives are often used to control whether the preprocessor should pass on to the compiler particular source code statements. In this particular case, we will address how to avoid having the preprocessor pass duplicate structure templates to the compiler. This is a common technique in TPF C.
Below we have the partial contents of a header file called **header1.h**. As you can see, it contains a structure template shown in blue. Also in blue, just above the template, is a #define statement. In red, above and below the structure template, we have two preprocessor conditional directives.

```
/* Partial contents of header1.h */

. . .
. . .

#ifndef name_structure

#define name_structure

struct name
{
     char last_name[25];
     char first_name[25];
};

#endif
```

```
. . .
. . .
```

The red conditional directives have been strategically placed in **header1.h** to control whether or not the code in blue is included in the compilation. What is the determining factor? It all depends upon whether or not `name_structure` referred to in the red conditional directive has been previously defined during the compilation of the source file that references this header file. The "if" condition of the `#ifndef name_structure` directive will be true if name_structure has ***not*** been defined.

If `name_structure` has ***not*** been previously defined, then all of the blue code in **header1.h** will be processed by the preprocessor and will be included in the source file. That means two things will happen: `name_structure` will be defined, by virtue of the `#define name_structure` statement, and the `struct name` template will be included in the source file that references this header file.

Now let's consider **header2.h** shown below...

```
/* Partial contents of header2.h */

. . .
. . .

#ifndef name_structure

#define name_structure

struct name
{
     char last_name[25];
     char first_name[25];
};

#endif



#ifndef address_structure

#define address_structure

struct address
{
     char street[80];
     char state[3];
};

#endif

. . .
. . .
```

The above **header2.h** file is very similar to the **header1.h** file we looked at before, except that this new header file contains two structure templates and associated preprocessor conditional directives, rather than just one.

Now, let's take this example a step further, so that we see the real benefit of placing the conditional directive logic around the structure templates. Let's suppose we want to include both the `struct name` and `struct address` templates in our **pcx1t3.c** source file. Well, we could simply have a **#include <heaader2.h>** statement in our **pcx1t3.c** program in order to get both templates in our source code. That's easy. However, let's say we also want some other things included that exist in **header1.h**. So we would include both header files, as we see in the box below...

```
/* Partial contents of pcx1t3.c */

. . .
. . .

#include <header1.h>

#include <header2.h>

. . .
. . .
```

This approach will work just fine, but it's important that you realize that it works because of the preprocessor conditional directives we've coded around our structure templates. If we remove the conditional directives, we will get a compilation error when we compile our **pcx1t3.c** program. Why? Because we are including two templates that are the same, namely the `struct name` template that exists in both **header1.h** and **header2.h**. The compiler will issue an error message telling us that our structure is being re-defined. That happens because the preprocessor includes ***both*** copies of the `struct name` template in the **pcx1t3.c** source code that's passed to the compiler.

By using the `#ifndef name_structure` directive, we avoid the compilation error. That's because when the preprocessor encounters the `#ifndef name_structure` directive in **header2.h**, the "if" condition will be false, because `name_structure` will already have been defined in **header1.h**. Because this condition fails, the preprocessor ***will not*** place a second copy of the `struct name` template into the source code to be compiled. Thus we avoid the compilation error.

The danger of including multiple copies of the same structure template is high in a complex environment like TPF C. For that reason, you will see these conditional directives frequently in TPF C header files. You need to "read" them, as well as how to code them.

The #error Preprocessor Directive

The **#error** preprocessor directive is used to produce a compile-time error message. The format of the directive is as follows:

```
#error error message
```

If the preprocessor's processing flow is such that a directive of this type is encountered by the preprocessor, then the specified error message will be displayed and the compilation will stop.
This directive is most useful for detecting a violation of a constraint during preprocessing. Consider the following example...

```
#ifdef CPLUS
#error C++ compiler required.
#endif
```

If indeed **CPLUS** had previously been defined during the compilation, then the #error directive would be processed by the preprocessor, because the **#ifdef** condition would be true. As a result, compilation would stop, and the "C++ compiler required." message would be displayed.

The #pragma Preprocessor Directive

The TPF implementation of C supports some features that are unique to TPF. The **#pragma** directive provides a means by which the compiler can offer machine or operating system specific features, while at the same time maintaining overall compatibility with C. However, these directives are not very portable
The format of the directive is as follows:

```
#pragma token-string
```

The *token-string* is a series of characters that provide an instruction to the compiler, with possible arguments. The pound or hash symbol (#) must be the first non-white-space character on the line containing the **#pragma** directive.
Below are some of the more common **#pragma** directive types...

#pragma map

You may see this form of the #pragma directive being used map or associate a TPF four character program name with a more meaningful name. For example...

```
#pragma map(convert_date,"UCDR")

...
...

convert_date();       /* really calling UCDR */
```

This allows for the calling of the **convert_date()** function, when really the TPF assembler program **UCDR** is being called.
However, you are encouraged to use a **#define** directive to give a more user-friendly alias to an external reference, rather than the #pragma map. In the past, Galileo has

used #pragma map statements for the same purpose. The #define statement is preferred over the #pragma since #pragmas are not portable. As an example...

```
#define CONVERT_DATE UCDR

...
...

CONVERT_DATE();        /* really calling UCDR */
```

#pragma pack

TPF assembler DSECTs are represented in TPF C as structures. By default a structure's data is unpacked , and, as such, the structure members align to natural boundaries, according to data type.
However, the #pragma pack directive can be used as a way of preventing natural boundary alignment, thereby eliminating unwanted pad bytes...

```
#pragma pack(1)
```

We'll discuss this directive in greater detail in the chapter that deals with converting DSECTs to structures.

#pragma title

This directive places the text specified after title onto all subsequent pages of the listing...

```
#pragma title("Example Program Number 1")
```

#pragma subtitle

This directive places the text specified after subtitle onto all subsequent pages of the listing. The subtitle will appear on the line following the title...

```
#pragma subtitle("First Section of the Program")
```

#pragma pagesize

This directive sets the number of lines for each page of the listing...

```
#pragma pagesize(50)
```

#pragma page

Skip the specified number of pages of the generated listing. If no number is provided, one page is skipped...

```
#pragma page(3)
```

#pragma skip

Skip the specified number of lines of the generated listing. The number must be a positive integer. The maximum is 254. If no number is provided, one line is skipped...

```
#pragma skip(200)
```

#pragma nomargins

Tells the TPF C compiler to use columns 1 through 80 are used for C Code. Normal compiles treat column 72 as the BAL Begin to Continue Column and columns 73 though 80 as sequence numbers. Most applications do not use sequence numbers and therefore will include the "#pragma nomargins". Systems programs do use sequence numbers and therefore will not use "#pragma nomargins".

```
#pragma nomargins
```

#pragma nosequence

Tells the TPF C compiler not to use sequence numbers. Normal compiles treat column 72 as the BAL Begin to Continue Column and columns 73 though 80 as sequence numbers. Most applications do not use sequence numbers and therefore will include the "#pragma nosequence". Systems programs do use sequence numbers and therefore will not use "#pragma nosequence".

```
#pragma nosequence
```

TPF C Header Files

The table below lists some of the most frequently used TPF C header files. A more complete list can be found in the IBM manuals provided through BookManager.
Please note that the "c$" prefix is being phased out, and a "c" suffix is being phased in.

Header File	Content
c$am0sg.h	Application message format (AMSG) structure
c$cinfc.h	Definitions for the cinfc() function
c$eb0eb.h	ECB structure
c$globz.h	Global tag definitions
c$miomi.h	TPF input message format
c$syseq.h	TPF system constants
sysapi.h	Restricted TPF API function prototypes and parameter definitions
tpfapi.h	TPF API function prototypes and constants
tpfeq.h	General TPF system function prototypes and constants
tpfglbl.h	Global interface function prototypes and constants
tpfio.h	TPF I/0 function prototypes and constants
tpflink.h	TPF linkage definitions (#pragma linkage statements; this file is required for compiling with the TARGET(TPF) option)
tpftape.h	TPF tape handling function prototypes and constants

TPF C Header File Relationships

TPF header files often contain #include directives that result in the including of other header files.
For example, the **tpfeq.h** header file contains a #include directive for the **c$eb0eb.h** header file. So if you have a #include for tpfeq.h in your program, then there is no need to explicitly include **c$eb0eb.h**.

TPF C Header File Naming Conventions

Read about Galileo's header file naming conventions on the intranet's ISO-C home page.

CHAPTER 8

Converting Existing DSECTs to Structures

Objectives

Upon completion of this chapter the participant will, with the aid of course materials, be able to:

- Convert a TPF assembler record DSECT into a TPF C structure
- Create a program containing a record structure template, along with the proper conditional preprocessor directives

Boundary Alignment Issues

When converting existing assembler DSECTs to C structures, you must take into account boundary alignment. (These is also true of the union data type.) Boundary alignment is determined by the way the different data types are stored. The following table shows the basic data types, how many bytes of storage each type takes up, and each type's natural boundary alignment in TPF:

Data Type	Memory Used	Boundary Alignment
char	I byte	byte
short	2 bytes	halfword
int	4 bytes	fullword
long	4 bytes	fullword
pointer	4 bytes	fullword
float	4 bytes	fullword
double	8 bytes	doubleword

Unpacked Structures

Under normal conditions, a structure will start on the strictest boundary required by any of its members, and each subsequent data element will be placed on its natural boundary as shown in the table above. This can result in pad bytes being added to the structure to accommodate natural boundary alignment. Such structures are considered to be "unpacked." Let's consider the following example...

```
...
...
struct example
{
     char name[3];
     short seat;
     float fare;
};
...
...
struct example passenger;
```

In the above example, storage for the **passenger** structure will be layed out according to the structure template and the rules of natural boundary alignment. As a result, the structure takes up twelve bytes of memory. The **name** member will take up three

bytes. The **seat** member takes up two bytes, but will naturally fall on a halfword boundary, so a pad byte will be allocated between **seat** and **name**. The **fare** member takes up four bytes, but will naturally fall on a fullword boundary, so two pad bytes will be allocated between **fare** and **seat**. In total, the structure takes up twelve bytes. The boundary alignment is based on the table above.

The addition of pad bytes is often undesirable in TPF, especially when we are creating a structure that is supposed to match an assembler DSECT, byte for byte. In order to get an exact match in terms of boundary alignment, we may need to "pack" the structure, as described below.

Packed Structures

To force a particular boundary alignment pattern we use the following **#pragma** directive...

```
#pragma pack(parameter) /* parameter values of 1, 2, 4 */
```

If you provide a parameter value of 1, then byte boundaries will be used in determining the aligment. This is the parameter value we will be specifying in this course.

If you provide a parameter value of 2, then halfword boundaries will be used in determining the aligment.

If you provide a parameter value of 4, then fullword boundaries will be used in determining the aligment. Providing no parameter value is the same as providing a parameter of 4.

Let's use the example we used above when we were talking about unpacked structures, but this time let's use the #pragma pack directive to force byte boundary alignment, so that there are no pad bytes between the structure members...

```
...
...
#pragma pack(1)        /* eliminate pad bytes */


struct example
{
     char name[3];
     short seat;
     float fare;
};
...
...
struct example passenger;
```

Because of the **#pragma pack(1)** directive, the structure will now take up only nine bytes of memory, instead of the twelve it would naturally take up without the directive. In other words, we have eliminated the pad bytes.

In constrast, if we were to force alignment by specifying **`#pragma pack(2)`**, the structure members would fall on halfword boundaries, and would take up a total of ten bytes of memory. This would cause a pad byte to exist between the **`name`** and **`seat`** members.

You should read the how to pack section of the ISO-C home page for the latest word on packing data.

You should also read the DSECT to C Header Conversion Standards document on Galileo's intranet ISO-C home page.

Examples of Converting a DSECT to a Structure

Example 1

In the table below, we are showing you typical assembler DSECT fields on the left side, and on the right side we show how those DSECT fields might be represented in TPF C. We say "might be represented" because converting an assembler field to a C field is not an exact science.
The C language equivalents in the table below are based on the size of the DSECT fields, as well as the nature of the data stored in the DSECT fields. Study each entry in the table, and make sure you understand how the conversion led to the TPF C equivalent declarations...

Assembler DSECT fields	TPF C language equivalents
`ABCBID&C DS H`	`char abcbid[2];`
`ABCCHK&C DS X`	`unsigned char abcchk;`
`ABCCTL&C DS X`	`unsigned char abcctl;`
`ABCPGM&C DS F`	`char abcpgm[4];`
`ABCFCH&C DS XL4`	`unsigned long abcfch;`
`ABCINX&C DS XL2`	`unsigned short abcinx;`
`ABCFLT&C DS XL4`	`unsigned long abcflt;`
`ABCITM&C DS 20CL55`	`struct abcitm abcitems[20];`

Notice that the first four conversions in the table are converted into TPF C fields of the **`char`** data type. Whenever you have a DSECT field of one byte in length, your choices for representation in TPF C are rather limited, because in C a one byte field must be either a character data type or a structure bit field data type. If you have no need to access individual bits in the one byte field, then character data type is most reasonable. That's why fields two and three in the table are represented as character

data types. If the original field is more than one byte in length, as it is for fields one and four in the table, then they are converted to an array of characters in TPF C.

The next three concern fields are used for storing data that is numberic in nature -- specifically a forward chain address, an index displacement, and a flight number. As a result, the corresponding TPF C fields are related to the integer data type (short or long).

The last field in the table refers to an item that is 55 bytes in length, of which there are twenty, repeated one after the other. This is well represented in TPF C as an array of twenty structures, with each structure being 55 bytes in length. We do not provide a view of the structure template here, but we should know that the size of the structure needs to be exactly 55 bytes, otherwise the structure will not match the size of an **ABCITM** in the original DSECT. Because of this fact, we will most likely need to use a **#pragma pack(1)** directive in our program to insure that the structure is packed (without pad bytes).

Example 2

Below you see assembler code that defines a DSECT for a theoretical **XX1PD** data macro, representing a record header of eighteen bytes, and twenty items of twentysix bytes each...

```
XX1BID&C DS  H           ID = XX
XX1CHK&C DS  X
XX1CTL&C DS  X
XX1PGM&C DS  F
XX1FCH&C DS  F
XX1INX&C DS  XL2        LOCATION INDEX:
XX1FLT&C DS  XL4        FLIGHT NUMBER:
XX1ITM&C DS  20CL26     ONE ITEM 26 CHARS 20 ITEMS
         ORG XX1ITM&C
XX1ADD&C DS  XL4
XX1PRC&C DS  X
XX1NAM&C DS  CL12
XX1BPT&C DS  CL3
XX1BDT&C DS  H
XX1TDT&C DS  H
XX1KRI&C DS  XL2
```

Below we show the DSECT as represented by three structure templates...

```
/* Define the xx1pd record */

#ifndef xx1pd

     #define xx1pd

     #pragma pack(1)       /*eliminate pad bytes */

     struct xx1_header    /* define header - 1st template */
     {
```

```
        unsigned char xx1bid[2];    /* basic ID */
        unsigned char xx1chk;       /* rcc       */
        unsigned char xx1ctl;       /* control byte */
        unsigned long xx1pgm;       /* pgm last filing */
        unsigned long xx1fch;       /* fwd chain addrs */
        unsigned short xx1inx;      /* Index for items */
        unsigned long xx1flt;       /* Flight Number */

    };



    struct xx1_item    /* define an item - 2nd template */
    {
        unsigned long xx1add;
        unsigned char xx1prc;
        unsigned char xx1nam[12];
        unsigned char xx1bpt[3];
        unsigned short xx1bdt;
        unsigned short xx1tdt;
        unsigned short xx1kri;
    };

    struct xx1pd          /* entire record - 3rd template
    */
    {
        struct xx1_header xx1_pdheader;

        struct xx1_item xx1pd_item[20];

    };

#endif
```

The following are important notes concerning the templates above...

1. The three templates collectively represent the DSECT: The first template (**`struct xx1_header`**) represents the record header, the second template (**`struct xx1_item`**) represents a record item, and the third template (**`struct xx1pd`**) specifies the first and second templates as its members. So, in other words, the first and second structures are nested members of the third structure.

   Notice that the second member of the third template is actually an array of structures -- specifically an array of twenty elements, each of which is an item structure that looks like the second template.

   We have chose to represent the DSECT in parts as three separate templates, rather than as one big template. This enables us to, in our TPF C code, easily refer to three different parts of the record... namely the header, the group of twenty items, and the individual items.

2. We used a **`#pragma pack(1)`** directive to insure that no pad bytes will be allocated when the record structure is declared, later in the program.

3. Notice that we surrounded the three templates with the preprocessor conditional directives **#ifndef xx1pd** and **#endif**. As we discussed earlier in the course, this is done to prevent the possibility of multiple inclusions of the templates. If **xx1pd** has not already been defined, then the **#define xx1pd** statement defines it, and all three templates get passed to the compiler. If, on the other hand, **xx1pd** has already been defined, then the **#define xx1pd** statement is not processed and the templates are not passed to the compiler.

4. Concerning the **#define xx1pd** directive, make sure you don't use the specified symbolic constant name (in this case, **xx1pd**) as a structure member name in your templates, as this will cause lots of problems when you compile.

5. If any of this is confusing to you, be sure to talk to a course manager about it. The use of structures in TPF C can be rather confusing, and yet it is very important that you understand how they work.

Considering Bit Fields

A one byte field in a DSECT that contains bit settings might look like this in the DSECT...

```
PD1ID3&C   DS     X
                     BIT 0  THIS ITEM IS CANCELLED FROM PNI.
                     BIT 1  GROUP BLOCK PNR
                     BIT 2  SOLD FROM GROUP BLOCK PNR
                     BIT 3  THIS IS A PTA PNR
                     BIT 4  NAME IS TRUNCATED
                     BIT 5  CORPORATE NAME
                     BIT 6  PRIMARY BIT
                     BIT 7  SUPPLEMENTARY INFO
```

A common way in which to represent that byte in TPF C would be as eight bit fields in a structure...

```
unsigned char pd1id3_bit0: 1;      /*cancelled */
unsigned char pd1id3_bit1: 1;     /*group block */
unsigned char pd1id3_bit2: 1;     /*sold from gb*/
unsigned char pd1id3_bit3: 1;     /*pta pnr */
unsigned char pd1id3_bit4: 1;     /*truncate nam*/
unsigned char pd1id3_bit5: 1;     /*corpe name */
unsigned char pd1id3_bit6: 1;     /*primary name*/
unsigned char pd1id3_bit7: 1;     /*fact/suppl */
```

The following is another option when you don't have an interest in showing the individual bits, and yet you do want to indicate that the data is in bit form...

```
unsigned char pd1id4_all_bits: 8;      /* indicators */
```

The bit fields should always be an **unsigned** data type, and depending upon your situation, you may wish to use **char**, **short**, or **long** to represent one byte, two bytes, or four bytes of bits, respectively.

CHAPTER 9

Accessing and Updating ECB Fields and Storage

Objectives

The purpose of this chapter is to introduce you to the Methods to access and update ECB fields. After completing this chapter, the student will be able to:

- Display an ECB Field.
- Update an ECB Field.
- Display a TPF Storage Field.
- Update a TPF Storage Field.

Accessing the ECB

In TPF C, the ECB is used as an interface to assembler segments and TPF itself. So, ECB Fields can be addressed and updated.
C language provides automatic storage in a stack area for each function. This does not violate the TPF re-entrancy principle. Programmers should take advantage of this automatic storage for transient data and work areas, instead of using the ECB.

struct eb0eb

struct eb0eb is the C equivalent of the assembler ECB dsect. It's defined in c$eb0eb.h, which is included tpfeq.h.

So, if you intend to use the ECB fields, you should #include <tpfeq.h> in your C Source Module.

ecbptr()

ecbptr() is a built-in function used to address the ECB. It returns a pointer to the ECB. When used with the arrow operator (->), the ECB fields can be accessed. Example: "ecbptr()->ebw000" would access the EBW000 field.

This "built-in" function resolves to a single instruction, so although it acts like a function, it's very efficient.

ECB

Another way to access the ECB is to dereference the pointer and use the dot (.) operator.
Example: "(*ecbptr()).ebw000" would access the EBW000 field.

This method is the Galileo International Preferred method, but it is a lot of typing. So, an alternative is to use the #define to set up a shorter tag. The logical tag is "ECB". So, the GI Preferred methodology would be:

```
#define ECB (*ecbptr())
...
ECB.ebw000 = 'a';
```

This method would also allow you to access the EBW000 field.

Coding Example of accessing and updating an ECB Field

```
/* This is a Test of accessing and updating an */
/* ECB field  */
/* Created for "Coding C for TPF" class */
#include <giFIRST.h>
#include <tpfeq.h>
#include <tpfapi.h>
#include <giTPF.h>

#define ECB (*ecbptr())

void pgm1 (void)
{
  ECB.ebw000 = 'A';

  if(ECB.ce1cr0 == 0)

  {

     printf("NO INPUT MESSAGE\n");

  }

  ecbptr()->ebx040 = 0;
  ecbptr()->ebx041 = 25;

  printf("ecbptr()->ebw000 = %c\n", ecbptr()->ebw000);
  printf("ECB.ebx040 = %hd\n", ECB.ebx040);

  exit (0);
}
```

Accessing a TPF Storage Field

First, you normally would define a structure or use a predefined structure for the TPF data that you are going to access. See chapter 8 for how to Convert a DSECT to a C Structure.
Then, you must get access to the address of the core block stored on the core level. If you use the find functions, you can assign the return value to your pointer. However, if the block has already been retrieved, you will have to use the core address stored in the ECB CBRW.

```
Example:  input_ptr = ecbptr->ce1cr0;
or        input_ptr = ECB.ce1cr0;
                      (if you have #defined ECB)
```

Coding Example of accessing and updating a TPF Storage Field

```
/* This is a Test of accessing and updating a */
/* TPF Storage field  */
/* Created for "Coding C for TPF" class */
#include <giFIRST.h>
#include <tpfeq.h>
#include <tpfapi.h>
#include <giTPF.h>
#define ECB (*ecbptr())

void pgm1 (void)
{
  char * input_ptr;

input_ptr = ECB.ce1cr0 + 20; /* point into the imsg */

  *input_ptr = 'A';  /* Update the TPF Storage block */

  printf("*input_ptr = %c\n", *input_ptr);

  exit (0);
}
```

CHAPTER 10

TPF C Input and Output Processing

Objectives

The purpose of this chapter is to introduce you to the TPF C Input and Output Functions. After completing this chapter, the student will be able to code:

- Function Calls for Input (scanf, sscanf and gets).
- Function Calls for Output (printf, sprintf, FMSG functions and puts).

The TPF C Functions for Reading Input from the Input Message Block

Because TPF is not an interactive environment, all terminal input is still the Input Message Block (MI0MI). You must now begin working with it in TPF C (c$mi0mi). The good news is that TPF C standard I/O processing causes functions gets(), scanf() and sscanf() to use the Input Message Block, without the need to specifically reference c$mi0mi.
The table below describes the primary TPF C functions related to reading TPF input messages.
Each C function in the table is a link to a section farther down on this page that describes the function in detail. Click on each function name to jump to that function's corresponding description, and then read that material so that you understand the function's purpose and syntax.

C Function	Assembler macro equivalent	Purpose
gets()	IMSG Editors	Read a string of input from the Input Message Block.
scanf()	Various Macros	Read message from the Input Message Block into specified variables.
sscanf()	Various Macros	Read message from the Input Message Block into specified variables.

Below, let's examine each of the listed TPF C functions...

gets() - Read a string of input from the Input Message Block

gets() requires a string location (user defined buffer) to move the input message to. It will read the data from level 0 in the ECB and move it the buffer. Once data on level 0 has been read, it cannot be re-read.
gets() returns a pointer to the string (data) that has been moved.

Include	stdio.h
Prototype	char * gets(char * ***buffer***); This function returns a pointer to the character string input. If an error occurs, or no data is read, then NULL is returned.
Parameter	***buffer*** is a pointer to an area to hold the line of text. The area must be large enough to accommodate the expected text.
Example	`#include <stdio.h>` `...` `char buffer[80], *bufptr;` `...` `if ((bufptr = gets(buffer))== NULL)` `{` `puts("NO INPUT FROM USER");` `}` `else` `{` `/* Process input */` `...` `}`
Usage Notes	1. gets() is a standard C library foutine that has been rewritten for TPF. 2. The '\0' is automatically placed at the end of the line in place of the EOL ('\n') or EOM character. 3. Data block on level 0 is used as the search point for the input data. 4. If the input consists of multiple lines, gets() should be

	repeated until NULL is returned. 5. TPF controls the way data is read. TPF bumps the gets() input pointer past the data that has already been retrieved, so once a gets() has been performed on a single line of input, it cannot be repeated to retrieve the same message. The second read will return a NULL. This is similar to what happens in the PC environment. 6. gets() will overlay any data in the specified buffer.

scanf() - Read message from the Input Message Block into specified variables

Works like the PC environment (except TPF is not interactive), once data on level 0 has been read, it cannot be re-read.
The format of the function call is the same in TPF as it is in the PC environment. The first parameter is the control string containing one or more conversion specifiers (%...). Following the control string are the addresses of variables to which the scanned values will be assigned. Each parameter must be seperated by a comma.
The return value from scanf is the integer number of successful assignments.

Include	stdio.h
Prototype	int scanf(const char * ***format***, **...**); This function returns the number of successful assignments.
Parameters	1. ***format*** is the control string that contains one or more conversion specifiers. 2. **...** represents one or more pointers to variables that are to be assigned values as a result of the scan process.
Example	`#include <stdio.h>` `...` `char board[3], off[3], month[3];` `short int date, count;` `...` `/* Input is A22JANLONPAR */` `...` `count = (short) scanf("A%hd%3c%3c%3c",` `&date, month, board, off);` `if (count < 4)` `{` `puts("ERROR IN MESSAGE");` `}` `else` `{` `/* Process input */` `...` `}`
Usage Notes	1. scanf() will ignore the characters in the control string that are not replacement values. In the above

	example the 'A' of the entry is ignored and the rest of the input is assigned to variables. 2. scanf() reads the message from the IMSG block on level 0. It then scans the input as specified by the format and sets up the variables listed. 3. scanf() uses the gets function, so you shouldn't mix the their use when dealing with the same input message. Otherwise you may get unpredictable results. 4. TPF controls the way data is read. TPF bumps the scanf() input pointer past the data that has already been retrieved, so once a scanf() has been performed on a single line of input, it cannot be repeated to retrieve the same message. The second read will return a NULL. This is similar to what happens in the PC environment. 5. Scanning is case sensitive. So, if you specify a portion of the input message to ignore, you should use upper case characters.

sscanf() - Read message from the Input Message Block into specified variables

The format of the function call is the same in TPF as it is in the PC environment. The first parameter is the control string containing one or more conversion specifiers (%...). Following the control string are the addresses of variables to which the scanned values will be assigned. Each parameter must be seperated by a comma.
sscanf() requires a user defined buffer where the input message is stored. It will read the data from the buffer and move the data into the specified variables. Unlike scanf(), sscanf() will allow you to re-read the same data more than once.
The return value from sscanf() is the integer number of successful assignments. sscanf() is great for re-reading the input message. If you have a message that is variable, you can use sscanf() until you find the correct format.

Include	stdio.h
Prototype	int sscanf(const char * ***buffer***, const char * ***format***, **...**); This function returns the number of successful assignments.
Parameters	1. ***buffer*** is a pointer to an area to scan from. The area must be large enough to accommodate the expected text. 2. ***format*** is the control string that contains one or more conversion specifiers. 3. **...** represents one or more pointers to variables that are to be assigned values as a result of the scan process.
Example	`#include <stdio.h>` `#define FALSE 0` `#define TRUE 1`

	`` ... char buffer[80]; char board[3], off[3], month[3]; short int date, count, bad_img = FALSE; ... /* Input is A22JANLONPAR or ALONPAR */ ... gets(buffer); count = (short) sscanf(buffer,"A%hd%3c%3c%3c", &date, month, board, off); if (count < 4) { count = (short) sscanf(buffer,"A%3c%3c", board, off); if (count < 2) { bad_img = TRUE; } } if (bad_img) { puts("ERROR IN MESSAGE"); } else { /* Process input */ ... } ```
Usage Notes	1. sscanf() will ignore the characters in the control string that are not replacement values. In the above example the 'A' of the entry is ignored and the rest of the input is assigned to variables. 2. sscanf() reads the message from the IMSG from the buffer. It then scans the input as specified by the format and sets up the variables listed. 3. Scanning is case sensitive. So, if you specify a portion of the input message to ignore, you should use upper case characters.

The Standard TPF C Functions for Sending Output to the Terminal

TPF is not an interactive environment, so sending output to the terminal is a little different that the PC environment.
For one, when you send output to the terminal, you will release the agent's keyboard. The agent then assumes that they can make another entry. So, if you send output to the terminal several times, you will confuse the agent at best, or even worse, wipe out their next entry. Therefore, in TPF you need to restrict the number of times that you send information to the terminal.
There are several ways to accomplish this. One, you can use sprintf() to build a buffer and then use printf() to send the buffer (when it is completed). Or, you can use FMSG to send the message.
The Galileo International preferred method is to use standard C functions to send data (printf(), sprintf() or puts()). Therefore you should try to use printf(), sprintf() and puts() as much as possible. However, there are draw backs to printf(), sprintf() and

puts(). One is that they all use the assembler WTOPC macro to send data. This creates a problem. WTOPC only allows up to 256 characters of data per macro. The other problem is that WTOPC will not use the automatic scrolling package to create a nice, neat screen. The other problem that has occurred with printf() is that it leaves the TPF "AAA in use" indicator on.
The printf(), puts() and sprintf() functions are currently undergoing a rewrite. In the near future, they will use FMSG to send the output message. Until then, you need to understand the WTOPC problem.
The problem with FMSG is that it is not standard C, and therefore it is discouraged at Galileo International.
The advantage of FMSG is that it doesn't limit you to 256 characters, and it uses the automatic scrolling package if needed.
This page will concentrate on printf(), sprintf() and puts(). For more information about FMSG, see the FMSG functions page.
The table below describes the primary TPF C functions related to sending TPF output messages.
Each C function in the table is a link to a section farther down on this page that describes the function in detail. Click on each function name to jump to that function's corresponding description, and then read that material so that you understand the function's purpose and syntax.

C Function	Assembler macro equivalent	Purpose
puts()	Various Macros	Output a character string to the terminal
printf()	Various Macros	Output formatted data to the terminal
sprintf()	Various Macros	Build formatted output data in a specified buffer

Below, let's examine each of the above TPF C functions...

puts() - Output a character string to the terminal

Include	stdio.h
Prototype	void puts(const char * ***string***); This function returns nothing.
Parameters	***string*** is a pointer to a line of text containing the output. The string must be terminated by a NULL (\0).

Example	`#include <stdio.h>` `...` `char buffer[80], *bufptr;` `...` `if ((bufptr = gets(buffer))== NULL)` `{` `puts("NO INPUT FROM USER");` `}` `else` `{` `/* Process input */` `...` `}`
Usage Notes	1. puts() is a standard C library routine that has been rewritten for TPF. 2. The NULL (\0) character must be placed at the end of the string to insure the data to be output. 3. The EBROUT field (containing the Terminal Address - LNIATA) must be set up with a valid address. If the entry was created via a create function (cremc, etc.) EBROUT will have to be established by the programmer. 4. puts() has been implemented at Galileo International with a call to WTOPC

printf() - Output formatted data to the terminal

Include	stdio.h
Prototype	int printf(const char * ***format***, **...**); This function returns the number of characters output.
Parameter	1. ***format*** is a pointer to a character string that is used to control the output format. 2. **...** represents one or more variables containing values to be output.

Example	`#include <stdio.h>` `...` `char board[3], off[3], month[3];` `short int date, count;` `...` `/* Input is A22JANLONPAR */` `...` `count = (short) scanf("A%hd%3c%3c%3c",` `&date, month, board, off);` `if (count < 4)` `{` `printf("ONLY %hd INPUTS WERE RECIEVED.\n",` `count);` `}` `else` `{` `printf("VALID MESSAGE");` `/* Process input */` `...` `}`
Usage Notes	1. printf() uses puts to output the data. Since puts uses WTOPC, the same restrictions/draw backs apply to printf. 2. printf() uses level F to build the data to be output. If a block is held on that level, a system error will be generated.

sprintf() - Build formatted output data in a specified buffer

sprintf() will move formatted data to a buffer. This prevents unlocking the agent's terminal. When all data is formatted, you will use a printf() to send the data to the terminal.
When moving data to the buffer, don't forget to count the number of characters output and bump your pointer to the buffer by that amount. Otherwise you will overlay previous data.

Include	stdio.h
Prototype	int sprintf(char * ***buffer***, const char * ***format***, **...**); This function returns the number of characters output.
Parameter	1. ***buffer*** is a pointer to the location to begin printing the data. 2. ***format*** is a pointer to a character string that is used to control the output format. 3. **...** represents one or more variables containing values to be output.

Example	``` #include <stdio.h> ... char out_buffer[256]; char board[3], off[3], month[3]; short int date, count; ... /* Input is A22JANLONPAR */ ... scanf("A%hd%3c%3c%3c",&date, month, board, off); count = sprintf(out_buffer,"THE DATE IS %hd.\n", date); count += sprintf(out_buffer+count,"THIS IS THE 2ND" "PART OF THE MESSAGE\n"); sprintf(out_buffer+count,"THE REST OF THE MESSAGE.\n"); printf("%s", out_buffer); ... ```
Usage Notes	1. sprintf() is used to build a multi-part message to be output to the terminal. It allows the building of the message without actually sending each part to the terminal individually. 2. You must maintain an accurate count of the characters already sent by sprintf(). So, use a variable to capture the return value of sprintf() (number of characters printed). On the 2nd through nth (last - 1) call to sprintf(), use the += operator to insure the variable is updated properly. On the last call to sprintf(), you don't need to keep track of the count anymore. 3. You must maintain an accurate beginning of where you want the print to begin. The first call to sprintf() will allow you to use the beginning of your buffer. Subsequent calls will require you to tell sprintf() where to begin, usually a buffer plus the number of previously printed characters (buffer+count). If you use buffer without adding a count, on subsequent calls to sprintf(), you will overwrite what you have already written. 4. When the message has been completed, use printf() to print the buffer as a string.

The TPF C FMSG Functions, Used for Sending Output to the Terminal

TPF is not an interactive environment, so sending output to the terminal is a little different that the PC environment.
For one, when you send output to the terminal, you will release the agent's keyboard. The agent then assumes that he/she can make another entry. So, if you

send output to the terminal several times, you will confuse the agent (at best) or even worse, wipe out their next entry.
Therefore in TPF, you need to restrict the number of times that you send information to the terminal.
Good news, there are two ways to accomplish this. One, you can use sprintf to build a buffer and then use printf to send the buffer (when it is completed). Or, you can use FMSG to send the message.
The Galileo International preferred method is to use standard C functions to send data (printf, sprintf or puts). Therefore, you should try to use printf, sprintf and puts as much as possible.
There are draw backs to printf, sprintf and puts. One is that they all use the Assembler WTOPC Macro to send data. This creates a problem. WTOPC only allows up to 256 characters of data per Macro. The other problem is that WTOPC will not use the automatic scrolling package to create a nice neat screen. The other problem that has occurred with printf is that printf leaves the TPF AAA INUSE indicator on.
The printf, puts and sprintf functions are currently undergoing a rewrite. In the near future, they will use FMSG to send the output message. Until then, you need to understand the WTOPC problem.
The problem with FMSG is that it is not standard C, therefore it is discouraged at Galileo International.
The advantage of FMSG is that it doesn't limit you to 256 and it uses the automatic scrolling package if needed.
This page will concentrate on the FMSG functions. For more information about printf, sprintf and puts, see the Sending Output page.
Normally, it is recommended to use sprintf to build a buffer and then use printf to send the buffer to the terminal. However, in light of the problems encountered with printf, many people are using sprintf to build the buffer and then using FMSG_SEND to send the buffer to the terminal.

Using Special FMSG function like Macros

There are several FMSG macros that look and work like function calls. See Chapter 7, #define for more information about function like macros. These FMSG macros were written for the old Target TPF (outdated C language in TPF). They have been updated to work with TPF C.
These FMSG function like macros are listed here because you may see them in existing TPF C code. Just remember that printf, puts and sprintf are the preferred functions when dealing with output.
Further information is available about FMSG function like macros in Tech Memo 1560. Written by Bryan Karr, March 1, 1994.
The table below describes the four TPF C functions related to using FMSG to send TPF output messages.
Each C function in the table is a link to a section farther down on this page that describes the function in detail. Click on each function name to jump to that function's corresponding description, and then read that material so that you understand the function's purpose and syntax.

C Function like Macro	Assembler macro equivalent	Purpose
FMSG_ADD()	FMSGI	Add a line of output, but don't send it

FMSG_N_ADD()	FMSG	Add a line of specified length to output, but don't send it
FMSG_SEND()	FMSG	Add a line of output and send it to the Terminal
FMSG_N_SEND()	FMSG	Add a line of output, set screen formatting options and send the output to the Terminal

Below, let's examine each of the above TPF C function like macros...

FMSG_ADD() - Add a line of output, but don't send it

Include	fmsgc.h
Prototype	void FMSG_ADD(***string***); This function returns nothing.
Parameters	***string*** is a pointer to a string. The string must be terminated by a NULL (\0) character.
Example	`#include <fmsgc.h>` `...` `char output_ln[65] = "This is an output message";` `...` **`FMSG_ADD(output_ln);`** `...`
Usage Notes	Data level 2 must be available.

FMSG_N_ADD() - Add a line of specified length to output, but don't send it

Include	fmsgc.h
Prototype	void FMSG_N_ADD(***length***, ***string***); This function returns nothing.
Parameter	1. ***length*** is the length of the output line that you wish to send to FMSG. This length should not include the Null (/0) character. 2. ***string*** is a pointer to a string. The string should be terminated by a NULL (\0) character, but because of the length argument, it is not necessary.
Example	`#include <fmsgc.h>` `...` `char output_ln[65] = "This is an output message";`

	... **FMSG_N_ADD(22, output_ln);** /* send "This is an output mess" */ ...
Usage Notes	Data level 2 must be available.

FMSG_SEND() - Add a line of output and send it to the Terminal

Include	fmsgc.h
Prototype	void FMSG_SEND(***line***, ***screen***, ***string***); This function returns nothing.
Parameter	1. ***line*** is the print line control character. Use: PRT_CURRENT to print on the current line. PRT_NEXT to print on the next line. PRT_OVERWRITE to overwrite the current line. 2. ***screen*** is the screen line control character. Use: SCREEN_TOP to clear the screen and start the output at the top of the screen SCREEN_CURRENT to start the output at the end of the current line. SCREEN_NEXT to start the output on the next line. 3. ***string*** is a pointer to a string. The string must be terminated by a NULL (\0) character.
Example	#include <fmsgc.h> ... char output_ln[65] = "This is an output message"; ... **FMSG_SEND(PRT_NEXT, SCREEN_TOP, output_ln);** /* send output to top of screen */ ...
Usage Notes	1. Data level 2 must be available. 2. FMSG will exit via UIO. Expect no return.

FMSG_N_SEND() - Add a line of output, set screen formatting options and send the output to the Terminal

Include	fmsgc.h
Prototype	void FMSG_N_SEND(***line***, ***screen***, ***length***, ***string***); This function returns nothing.

Parameter	1. ***line*** is the print line control character. Use: PRT_CURRENT to print on the current line. PRT_NEXT to print on the next line. PRT_OVERWRITE to overwrite the current line. 2. ***screen*** is the screen line control character. Use: SCREEN_TOP to clear the screen and start the output at the top of the screen SCREEN_CURRENT to start the output at the end of the current line. SCREEN_NEXT to start the output on the next line. 3. ***length*** is the length of the output line that you wish to send to FMSG. This length should not include the Null (/0) character. 4. ***string*** is a pointer to a string. The string must be terminated by a NULL (\0) character.
Example	`#include <fmsgc.h>` `...` `char output_ln[65] = "This is an output message";` `...` **`FMSG_SEND(PRT_NEXT, SCREEN_TOP, 22, output_ln);`** `/* send "This is an output mess"` `to top of screen */` `...`
Usage Notes	1. Data level 2 must be available. 2. FMSG will exit via UIO. Expect no return.

CHAPTER 11

TPF C Storage Functions

Objectives

Upon completion of this chapter the participant will, with the aid of course materials, be able to:

- Use the working storage functions of TPF C
- Use the heap storage functions of TPF C

TPF C Working Storage Functions

The table below describes the primary TPF C functions related to managing TPF working storage.
Each C function in the table is a link to a section farther down on this page that describes the function in detail. Click on each function name to jump to that function's corresponding description, and then read that material so that you understand the function's purpose and syntax.

C Function	Assembler macro equivalent	Purpose
getcc()	GETCC	Obtain a working storage block
relcc()	RELCC	Release a working storage block
detac()	DETAC	Detach a working storage block
attac()	ATTAC	Reattach a detached working storage block
flipc()	FLIPC	Flip two specified CBRWs
levtest()	LEVTA	Test a specified data level
crusa()	CRUSA	Test and release working storage blocks

Below, let's examine each of the above TPF C functions...

getcc() - Obtain a Working Storage Block

Include	tpfapi.h
Prototypes	void * getcc(enum t_lvl ***level***, enum t_getfmt ***format***, ***spec1***); void * getcc(enum t_lvl ***level***, enum t_getfmt ***format***, ***spec1***, ***spec2***);

	The function returns a void pointer to the working storage block obtained.
Parameters	1. ***level*** is a valid data level (D0 - DF). 2. ***format*** must specify at least one of the following to determine how storage is allocated: **GETCC_TYPE** is used when specifying a logical block type (L0, L1, L2 or L4) as spec1. **GETCC_SIZE Spec1** would be the size of storage required. The smallest block size that can satisfy the request will be used. Asking for more than 4095 bytes results in SERRC with exit. **GETCC_ATTR0 -> ATTR9, GETCC PRIME or GETCC_OVERFLOW** Use one of the these options to specify a 2 character record ID from the RIAT. Prime and Overflow options are supported for migration only and relate to TYPE0 and TYPE1. In addition the 2 options below can be coded: **GETCC_FILL Spec2** would specify the fill character to initialize the block. By default all blocks will be initialized to binary zeros. **GETCC_COMMON** is used to obtain common storage - the default is shared storage. 3. ***spec1*** varies depending on the value of the ***format*** parameter: If ***format*** is **GETCC_TYPE**, then ***spec1*** must contain a logical block size (LO, L1, L2 or L4). If ***format*** is **GETCC_SIZE**, then ***spec1*** must contain the number of bytes required (maximum 4095). If ***format*** is one of the **GETCC_ATTR..** options, then ***spec1*** must contain a 2-character record ID in double quotes. 4. ***spec2*** is provided only if **GETCC_FILL** is specified. ***spec2*** will then be the hex value with which the core block is to be initialized.
Example	`#include <tpfapi.h>` `. . .` `struct pd1pd *pd1pd_ptr;` `char * blk1ptr, * blk2ptr, * blk3ptr;` `. . .` `. . .` `pd1pd_ptr = getcc(D3,GETCC_ATTR1,"PD");`

	`blk1ptr = getcc(D4,GETCC_TYPE,L2);` `blk2ptr = getcc(D5,GETCC_SIZE+GETCC_FILL,400,' ');` `blk3ptr = getcc(D6,GETCC_TYPE+GETCC_COMMON,L4);` `. . .` `. . .`
Usage Notes	1. The level specified must not be holding a block. 2. **GETCC_COMMON** requires runtime authorisation. 3. **GETCC_PRIME** and **GETCC_OVERFLOW** are used for migration purposes. For non-migration purposes, use **GETCC_ATTR0** and **GETCC_ATTR1** for prime and overflow, respectively. 4. A system error with exit will occur if: • Size of requested storage is greater than 4095. • Record ID is invalid.

relcc() - Release a Working Storage Block

Include	tpfapi.h
Prototype	void relcc(enum t_lvl ***level***); This function does not return a value. It causes the working storage block on the specified level to be returned to the system for re-use.
Parameter	***level*** is a valid data level (D0 - DF).
Example	`#include <tpfapi.h>` `. . .` `. . .` `relcc(D2);` `. . .` `. . .`
Usage Notes	A system error with exit will occur if the specified level is not holding a block.

detac() - Detach a Working Storage Block

Include	tpfapi.h
Prototype	void detac(enum t_lvl ***level***); The function returns nothing. The core block reference word for the specified level is modified to indicate that no block is held.
Parameter	***level*** is a valid data level (D0 - DF).
Example	`#include <tpfapi.h>` `. . .` `. . .` **`detac(D2);`** `. . .` `. . .`
Usage Notes	A system error with exit will occur if the specified level is not holding a block.

attac() - Reattach a Detached Working Storage Block

Include	tpfapi.h
Prototype	void * attac(enum t_lvl ***level***); Returns a void pointer to the address of the working storage block that was re-attached.
Parameter	***level*** is a valid data level (D0 - DF).
Example	`#include <tpfapi.h>` `#include "c$pd1pd.h"` `struct pd1pd *pd1ptr;` `. . .` `. . .` `pd1ptr = `**`attac(D2);`** `. . .` `. . .`
Usage Notes	1. A system error with exit will occur if: - The specified level is holding a block - No storage was previously detached from the specified level

	2. The most recently detached block is attached first.

flipc() - Swap CBRW's and FARW's for specified Levels

Include	tpfapi.h
Prototype	void flipc(enum t_lvl ***level1***, enum t_lvl ***level2***); The function returns nothing. The core block reference words and file address reference words for the specified levels are interchanged.
Parameters	1. ***level1*** is a valid data level (D0 - DF). 2. ***level2*** is a valid data level (D0 - DF).
Example	`#include <tpfapi.h>` `. . .` `. . .` **`flipc(D2,D3);`** `. . .` `. . .`
Usage Notes	1. Specifying the same data level for both parameters results in no action being taken. 2. Neither level should have an I/O operation in progress, as the status information in the ECB is unknown until a subsequent waitc().

levtest() - Test for a Working Storage Block on a Level

Include	tpfapi.h
Prot otype	int levtest(enum t_lvl ***level***); The function returns an integer. If the level is in use, the integer will represent the size of the block on the level tested. If the level tested is not in use, a value of zero will be returned.
Parameter	***level*** is a valid data level (D0 - DF).
Example	`#include <tpfapi.h>` `. . .` `. . .`

	char * block_ptr; . . . if (**levtest(D2)**== 0) /* if level is free */ block_ptr = getcc(D2,GETCC_TYPE,L4); else block_ptr = ecbptr()->ebccr2; . . . . . .
Usage Notes	None

crusa() - Release Working Storage Block(s) if held

Include	tpfapi.h
Prototype	void crusa(int ***count***, enum t_lvl ***level***, ***...***); The function returns nothing. Core block reference words on specified level(s) will be updated if a block is released.
Parameters	1. ***count*** is the number of levels to be checked 2. ***level*** is a valid data level (D0 - DF). 3. ***...*** represents additional levels.
Example	#include <tpfapi.h> . . . . . . **crusa(3,D2,D4,D6);** . . . . . .
Usage Notes	A system error with exit will occur if ***count*** is incorrect or invalid.

TPF C Heap Storage Functions

The table below describes the primary TPF C functions related to managing TPF heap storage.
Each C function in the table is a link to a section farther down on this page that describes the function in detail. Click on each function name to jump to that function's corresponding description, and then read that material so that you understand the function's purpose and syntax.

C Function	**Assembler macro equivalent**	**Purpose**
malloc()	MALOC	Allocate space in heap storage
calloc()	CALOC	Allocate initialized space in heap storage

realloc()	RALOC	Reallocate size of space in heap storage
free()	FREEC	Deallocate space in heap storage

Heap storage allows an application to request a unlimited amount of contiguous bytes of storage. Only programs that are allocated as 31 bit can use heap storage (C programs must be 31 bit). Frames allocated to an ECB to satisfy requests for heap storage remain attached to the ECB until exit.
Data levels are not involved in the management of heap storage. Keeping track of the start address given is the responsibility of the application (via pointers).

Below, let's examine each of the above TPF C functions...

malloc() - Allocate Heap Storage

Include	stdlib.h
Prototype	void * malloc(size_t ***size***); The function returns a void pointer to the allocated space. If no space is allocated, NULL is returned.
Parameter	***size*** is an integer value of the number of bytes requested.
Example	`#include <stdlib.h>` `. . .` `. . .` `struct pas * pass` `. . .` `/* this example allocated dynamically */` `/* variable count = number of passengers */` `. . .` `pass = (struct pas *) malloc((sizeof(struct pas)* count))` `if (pass == NULL)` `{` `  puts ("malloc error");` `}` `else` `{` `  ...valid processing...` `}`
Usage Notes	1. Memory allocated as a result of coding a malloc() is assigned from heap storage. 2. When malloc() is successful in allocating heap storage, the value returned by by the function is a pointer to void. In other words, the starting address of the assigned heap storage is returned. Because the returned address is of the "pointer to void" data type, it is customary to explicitly cast the returned address to match the data type of the pointer being assigned the returned address. This practice is shown in the example above. The **pass** pointer that the returned address is being assigned to is of the **struct pas *** data type, so we explicitly cast the returned address by

	providing **(struct pas *)** in front of the malloc() call.

calloc() - Allocate and Initialize Heap Storage for an Array

Include	stdlib.h
Prototype	void *calloc(size_t ***num***, size_t ***size***); The function returns a void pointer to the allocated space. If no space is allocated, NULL is returned.
Parameter	***num*** is an integer value of the number of objects in array. ***size*** is an integer value of the number of bytes in each element of array.
Example	`#include <stdlib.h>` `. . .` `struct pas * pass` `. . .` `/* array to deal with 50 passengers */` `. . .` `pass = (struct pas *) calloc(50,(sizeof(struct pas)));` `. . .` `. . .`
Usage Notes	1. All space is initialized to binary zeros. 2. Memory allocated as a result of coding a calloc() is assigned from heap storage. 3. When calloc() is successful in allocating heap storage, the value returned by by the function is a pointer to void. In other words, the starting address of the assigned heap storage is returned. Because the returned address is of the "pointer to void" data type, it is customary to explicitly cast the returned address to match the data type of the pointer being assigned the returned address. This practice is shown in the example above. The **pass** pointer that the returned address is being assigned to is of the **struct pas *** data type, so we explicitly cast the returned address by providing **(struct pas *)** in front of the calloc() call.

realloc() - Change Size of Previously Allocated Heap Storage

Include	stdlib.h

Prototype	void *realloc(void * ***ptr***, size_t ***size***); The function returns a void pointer to the allocated space. The original location may have changed. If no space is allocated or specified size was zero, NULL is returned.
Parameter	***ptr*** is a pointer to previously allocated storage. If ptr is NULL, realloc() behaves like malloc(). ***size*** is an integer value of the number of bytes to be allocated.
Example	`#include <stdlib.h>` `long * blkptr;` `. . .` `blkptr = (long *) malloc(100);` `. . .` `. . .` `blkptr = (long *)` **`realloc(blkptr, 200);`** `. . .` `. . .`
Usage Notes	1. The contents of the reallocated storage is unchanged up to the shorter of the new and old sizes. 2. When realloc() is successful in allocating heap storage, the value returned by by the function is a pointer to void. In other words, the starting address of the assigned heap storage is returned. Because the returned address is of the "pointer to void" data type, it is customary to explicitly cast the returned address to match the data type of the pointer being assigned the returned address. This practice is shown in the example above. The **blkptr** pointer that the returned address is being assigned to is of the **long *** data type, so we explicitly cast the returned address by providing **(long *)** in front of the realloc() call.

free() - Free Heap Storage

Include	stdlib.h
Prototype	void free(void * ***ptr***); The function returns nothing. It frees heap storage previously allocated using a malloc(), calloc(), or realloc().

Parameter	***ptr*** is a pointer to space previously allocated by malloc() or calloc().
Example	`#include <stdlib.h>` `. . .` `struct pas * pass` `. . .` `/* array to deal with 50 passengers */` `. . .` `pass = (struct pas *) calloc(50,(sizeof(struct pas)));` `. . .` **`free(pass);`** `. . .` `. . .`
Usage Notes	Be sure that the pointer parameter contains the correct address.

CHAPTER 12

Calls to Other Programs

Functions are the basic building blocks of C language.
There are three types of functions...

- External
- Static
- Library

The following matrix shows a summary of the calls that are supported in TPF C...

	External Functions	**Library Functions**	**Static Functions**	**Assembler Programs**
External Functions	YES	YES	YES*	YES
Library Functions	YES	YES	YES*	NO
Static Functions	YES	YES	YES*	YES
Assembler Programs	YES	NO	NO	YES

The * means from within the same DLM

Now let's consider some of the calls in greater detail. Proceed through the chapter by reading all sections in order, beginning with the chapter objectives.

Objectives

Upon completion of this chapter the participant will, with the aid of course materials, be able to:

- Pass data from a C program to an Assembler program using the TPF_regs structure
- Pass data from one C program to another C program

TPF C Programs Calling Other TPF C Programs

Every TPF C program must exist within a DLM.
The TPF C functions defined within the various TPF C programs contained in a DLM can be of two types:

- external functions
- static functions

The two types, external and static, actually refer to the storage class of the function.

When one TPF C program calls another TPF C program, the call is implemented as a function call.

External Functions

All functions are external functions by default, and can be called by any other function within the same DLM. In other words, their scope is global within the DLM.
An external function can also be called by functions in other DLMs if that external function is the entry point function of the DLM in which it resides.

Static Functions

Static functions are static by virtue of the fact that the keyword ***static*** is explicitly stated as part of the function definition and function prototype.
Static functions are known only within the program in which they are defined. In other words, their scope is local to a specific program.

TPF C Programs Calling Assembler Programs

TPF C functions can generally call assembler programs without any special calling protocol. In such cases, the run-time parameter linkage is handled automatically by the control program. (There may be some circumstances that are an exception to the general rule.)
When a TPF C function calls an assembler program, there must be, in the calling TPF C program, a valid C function prototype for the assembler program being called.

Using a TPF_regs Structure

When you wish to pass data to the called assembler segment, you have two choices:

- Pass data via the ECB and/or main storage
- Pass data via the address of a TPF_regs structure

TPF_regs refers to a TPF C structure template that resides in the **tpfregs.h** header file. The structure's members are of the **long int** data type, and they correspond to general registers R0 through R7. The members have the following names:

r0
r1
r2
r3
r4
r5
r6
r7

We can use this structure as a means of passing values to a called assembler segment. Here's how it works...

- Include the proper the **tpfregs.h** header file
- Provide a function prototype for the call to the assembler segment, with a **void** return type, and with the address of a **TPF_regs** structure as a parameter
- Declare a **TPF_regs** structure
- Assign integer values to the structure members
- Pass the adddress of the structure when calling the assembler segment

As a result, the values assigned to the members will be accessable to the assembler segment in its general registers that correspond with the structure members

Even though the prototype specifies a **void** return, the assembler program's registers R0 - R7 are returned to the C program in the TPF_regs structure.
If you do not wish to pass any values via the structure, then pass NULL pointer, instead.

TPF_regs Structure Example

The following code fragment shows a **TPF_regs** structure being used to pass three register values to the assembler program **FACS**, in order to obtain the file address of a **PNID** fixed file record...

```
#include "gifirst.h"            /* needed header files */
#include <tpfregs.h>
#include "gitpf.h"
#define RECTYPE "#PNDRI "        /* PNID record type */
void FACS(struct TPF_regs * regs_ptr);  /* FACS prototype */

struct TPF_regs regs;            /* declare structure */
struct TPF_regs * regs_ptr = &regs;

long ordnum = 50;
/* set r6,r7, and r0 with values FACS will need */
regs.r6 = (long) RECTYPE;
regs.r7 = (long) &(ecbptr()->ce1fa5); /* addrs of FARW on D5 */
regs.r0 = ordnum;
FACS(&regs);          /* Call FACS */
if(regs.r0 == 0)            /* Check for FACS failure */
{
   printf("Error on FACE");
```

```
   exit(0);
}
else
{
   ...success processing
}
. . .
. . .
```

TPF Assembler Programs Calling TPF C Programs

An assembler program segment can call a TPF C entry point function residing in a DLM. The assembler segment should call the TPF C program via an ENTRC macro, rather than an ENTNC or an ENTDC, because it is the nature of a TPF C program to return to the point of call.
You can, of course, pass data from the assembler segment to the C program via the ECB and main storage.
In addition, IBM has created a **restricted interface** for passing parameters from assembler to C, but IBM urges programmers to refrain from using the interface unless it is absolutely necessary, because the interface may change in future releases of TPF. In this introductory course, we will not go into the uses of the restricted interface.

CHAPTER 13

TPF C File Input and Output

This chapter concerns the TPF C functions used to access and update records stored on DASD.
Proceed through the chapter by reading all sections in order, beginning with the chapter objectives.

Objectives

Upon completion of this chapter, the participant will, with the aid of course materials, be able to:

- Code a TPF C file function to find a fixed file record on DASD and copy the record into a core block
- Code an TPF C file function to write a record to DASD after the record has been updated in core
- Code an TPF C pool file management function to obtain a pool address and accompanying working storage block for the purpose of creating a new chain record
- Write the code needed to chain a new pool record to a prime record, and file the necessary updates

Obtaining a Fixed File Address from FACS or FACE

The TPF assembler programs **FACS** and **FACE** are called for the purpose of obtaining the address of a fixed file record stored on DASD.
The table below describes the primary way in which to call the FACS or FACE programs. Using this method, we are making the call as we would to any other TPF assembler program, providing register values to FACS or FACE through the use of the **TPF_regs** structure.

Calling FACS or FACE to obtain a fixed file address

Include	tpfeq.h
Prototypes	void FACS(struct TPF_regs * ***regs***); void FACE(struct TPF_regs * ***regs***);
Parameter	***regs*** is a pointer to a struct TPF_regs structure. Prior to the call to FACS or FACE, the r0, r6 and r7 structure members must contain appropriate values, as described in the "setup" section below.
Setup	1. Set **R0** equal to the ordinal number of the record you are seeking the address of. 2. For **FACS**, set **R6** equal to the address of symbolic record type.

	For **FACE**, set **R6** equal to the address of numeric record type. 3. Set **R7** equal to the address of an 8-byte area where you want the returned file address to be placed.
Example	``` #include "gifirst.h" #include <tpfeq.h> #include "gitpf.h" #define PNID_RECTYPE "#PNDRI " /* Note that the record type value specified above */ /* must be in double quotes as eight characters, */ /* left justified and padded with blanks */ void FACS(struct TPF_regs * regs); /* prototype */ void pcx1(void) { long ordnum = 50; struct TPF_regs regs; . . . . . . regs.r6 = (long) PNID_RECTYPE; regs.r7 = (long) &(ecbptr()->ce1fa5); regs.r0 = ordnum; FACS(&regs); /* call FACS */ if (regs.r0 == 0) { if (regs.r7 == 1) { /* Invalid record type */ . . . . . . } else { /* Invalid ordinal number */ . . . . . . } } . . . . . . ```
Usage Notes	1. When setting R6 equal to the address of symbolic record type for a call to FACS, be sure to define the symbolic record type in double quotes as being eight characters in length, left justified and padded with blanks, as shown in the example above. 2. Upon return from FACS or FACE, when there are no FACS or FACE errors... • R0 will be set to the highest ordinal that exists for the specified record type.

	• File address will reside at location specified in setup. 3. Upon return when a FACS or FACE error takes place... • R0 will equal zero. • R7 will equal one if an invalid record type was provided in the call to FACS or FACE. • R7 will equal two if an invalid ordinal number was provided in the call to FACS or FACE.

The TPF C Functions for Reading Files from DASD

The table below describes the primary TPF C functions related to reading TPF records in to core from DASD.
Each C function in the table is a link to a section farther down on this page that describes the function in detail. Click on each function name to jump to that function's corresponding description, and then read that material so that you understand the function's purpose and syntax.

C Function	Assembler macro equivalent	Purpose
find_record()	FINWC FIWHC	Read a record from file with a wait, and possibly a hold
findc()	FINDC	Read from file
finwc()	FINWC	Read from file with a wait
finhc()	FINHC	Read from file with a hold
fiwhc()	FIWHC	Read from file with a wait and hold
waitc()	WAITC	Suspend processing until I/O completes

Below, let's examine each of the above TPF C functions...

find_record() - Read a record from file with a wait, and possibly a hold

Include	tpfio.h
Prototype	void * find_record(enum t_lvl ***level***, const unsigned int ****address***, const char * ***id***, unsigned char ***rcc***, enum t_act ***type***); The function returns address of the core block into which the record was copied. If an error occurs, the function returns a zero and sets the SUD indicator in the ECB.
Parameters	1. ***level*** is a valid data level (D0 - DF). 2. ***address*** is a pointer to the file address of the file to be retrieved. 3. ***id*** is a pointer to the 2-character record ID. 4. ***rcc*** is the 1-character record code check. 5. ***type*** is HOLD or NOHOLD, indicating whether record is retrieved with an exclusive hold.
Example	`#include "gifirst.h">` `#include <tpfio.h>` `#include "gitpf.h"` `struct somefile` `{` `  char file_id[2];` `  char file_rcc;` `  ...` `};` `void pcx1(void)` `{` `  struct somefile * file_ptr;` `  ...` `/* Note: This example requires file addrs in CE1FA5 */` `  file_ptr = find_record(D5,NULL,"SF",23,NOHOLD);` `  if (file_ptr == 0)` `  {` `  /* Process find error */` `  ...` `  }` `  ...`
Usage Notes	1. The file address reference word (FARW) of the specified level is initialized by this function, as well as the core block reference word upon return. 2. If the FARW already contains the correct address or ID, you can code NULL as those parameters. 3. The record ID and RCC parameters are compared to those of the record being retrieved. If either check fails, SUD indicators will be set. 4. To bypass the record ID check, code RECID_RESET

	as the ID parameter. 5. To bypass the RCC check, code '\0' as the RCC parameter. 6. A waitc is imbedded within this function. 7. The level specified must not be holding a block.

findc() - Read a record from file

Include	tpfio.h
Prototype	void findc(enum t_lvl ***level***); The function returns nothing.
Parameter	***level*** is a valid data level (D0 - DF).
Example	`#include "gifirst.h">` `#include <tpfio.h>` `#include "gitpf.h"` `struct somefile` `{` `  char file_id[2];` `  char file_rcc;` `  ...` `};` `void pcx1(void)` `{` `  struct somefile * file_ptr;` `  ...` `/* Note: This example requires file addrs,rec id,RCC are setup in CE1FA5 */` `  findc(D5);` `  if (waitc()!= 0)  /* findc() requires a waitc() */` `  {` `  /* Process find error */` `  ...` `  }` `  else  /* point to record on level 5 */` `  {` `  file_ptr = ecbptr()->ce1cr5;` `  ...` `  }` `  ...` `  ...`
Usage Notes	1. The file address reference word must contain the correct record ID, record code check and file address. If any of those parameters are incorrect, the find will fail, and the SUD indicators in the ECB will be set. 2. A waitc function call must be issued before accessing

	the record in core. 3. The function gets a core block, attaches it to the specified level, and updates the core block reference word. 4. The level specified must not be holding a block.

finwc() - Read a record from file with a wait

Include	tpfio.h
Prototype	void * finwc(enum t_lvl ***level***); The function returns address of the core block into which the record was copied. If an error occurs, the function returns a zero and sets the SUD indicator in the ECB.
Parameter	***level*** is a valid data level (D0 - DF).
Example	`#include "gifirst.h">` `#include <tpfio.h>` `#include "gitpf.h"` `struct somefile` `{` `  char file_id[2];` `  char file_rcc;` `  ...` `};` `void pcx1(void)` `{` `  struct somefile * file_ptr;` `  ...` **`/* Note: This example requires File addrs,rec id,RCC setup in CE1FA5 */`** `  file_ptr = finwc(D5);` `  if ( file_ptr == 0 )` `  {` `  /* Process find error */` `  ...` `  }` `  ...`
Usage Notes	1. The file address reference word must contain the correct record ID, record code check and file address. If any of those parameters are incorrect, the find will fail, and the SUD indicators in the ECB will be set. 2. The function gets a core block, attaches it to the specified level, and updates the core block reference word.

	3. A waitc is imbedded within this function. 4. The level specified must not be holding a block.

finhc() - Read a record from file with a hold

Include	tpfio.h
Prototype	void finhc(enum t_lvl ***level***); Returns nothing.
Parameter	***level*** is a valid data level (D0 - DF).
Example	`#include "gifirst.h">` `#include <tpfio.h>` `#include "gitpf.h"` `struct somefile` `{` `  char file_id[2];` `  char file_rcc;` `  ...` `};` `void pcx1(void)` `{` `  struct somefile * file_ptr;` `  ...` `/* Note: This example requires file addrs,rec id,RCC are setup in CE1FA5 */` `  finhc(D5);` `  if (waitc()!= 0)  /* finhc() requires a waitc() */` `  {` `  /* Process find error */` `  ...` `  }` `  else  /* point to record on level 5 */` `  {` `  file_ptr = ecbptr()->ce1cr5;` `  ...` `  }` `  ...`
Usage Notes	1. The file address reference word must contain the correct record ID, RCC and file address. If any of those parameters are incorrect, the find will fail, and the SUD indicators in the ECB will be set. 2. A waitc function call must be issued before accessing the record in core. 3. The function gets a core block, attaches it to the specified level, and updates the core block reference

	word. 4. The level specified must not be holding a block.

fiwhc() - Read a record from file with a wait and hold

Include	tpfio.h
Prototype	void * fiwhc(enum t_lvl ***level***); This function returns the address of the core block into which the record was copied. If an error occurs, the function returns a zero and sets the SUD indicator in the ECB.
Parameter	***level*** is a valid data level (D0 - DF).
Example	`#include "gifirst.h">` `#include <tpfio.h>` `#include "gitpf.h"` `struct somefile` `{` `  char file_id[2];` `  char file_rcc;` `  ...` `};` `void pcx1(void)` `{` `  struct somefile * file_ptr;` `  ...` **`/* Note: This example requires file addrs,rec id,RCC are setup in CE1FA5 */`** `  file_ptr = fiwhc(D5);` `  if ( file_ptr == 0 )` `  {` `  /* Process find error */` `  ...` `  }` `  ...`
Usage Notes	1. The file address reference word must contain the correct record ID, RCC and file address. If any of those parameters are incorrect, the find will fail, and the SUD indicators in the ECB will be set. 2. The function gets a core block, attaches it to the specified level, and updates the core block reference word.

waitc() - Suspend processing until I/O completes

Include	tpfio.h
Prototype	int waitc(void); This function returns a zero if no errors occur. Otherwise, returns an integer SUG value.
Parameters	None
Example	`#include "gifirst.h">` `#include <tpfio.h>` `#include "gitpf.h"` `struct somefile` `{` `  char file_id[2];` `  char file_rcc;` `  ...` `};` `void pcx1(void)` `{` `  struct somefile * file_ptr;` `  ...` **`/* Note: This example requires file addrs,rec id,RCC are setup in CE1FA5 and CE1FA6 */`** **`  findc(D5);`** **`  findc(D6);`** `  if (waitc()!= 0)` `  {` `  /* Process find error */` `  ...` `  }` `  ...` `  ...`
Usage Notes	If an error related to I/O hardware occurs, a dump is issued and a CRAS message is sent.

File Skill Check 1

For this exercise we are asking you to call the assembler program **FACS** to obtain the file address of a particular fixed file, prime PNID record, the ordinal of which will be provided as input from the user.
Once your program has obtained that file address, use an appropriate function (one of the ones discussed earlier in this chapter) to obtain a copy of the record and have it placed in core to be read. Additionally, use the PNID structure templates you created earlier in the course to facilitate accessing the fields in the record.
The record type of the PNID is #PNDRI (decimal equivalent is 12).
This exercise requires reading information from records, but you will not be updating any records. In other words, you will not be filing any records.

Use **TRB1xx** as your TPF C program name, with "**xx**" being the version you have been assigned by the course managers. You will need to load the driver program **TRA1T4**, and share **TRN0002**.
The course managers will provide you with several ordinal numbers to input in the beginning. However, you're free to use other PNID ordinals as well.

Input:

The user of your program will provide a specific PNID ordinal number as part of the input message. The user entry can follow either of these two formats:

t*trb1/***b ordinal#***

t*trb1/***c ordinal#***

The letter 'b' after t*trb1/ indicates that the user wants displayed a list of passenger names associated with all the **booked items** in the prime PNID record. (The prime only. Don't bother accessing records chained to the prime.)

The letter 'c' after t*trb1/ indicates that the user wants displayed a list of passenger names associated with all the **cancelled items** in the prime PNID record. (The prime only. Don't bother accessing records chained to the prime.)

In the input message, check for the slash(/), and the 'b' or 'c', and ensure that there are at least two arguments after the slash.

Output:

Use the sprintf() function to build at least part of your display. (You may wish to use puts() or printf() to display something short and simple, such as a "LIST OF BOOKED PASSENGERS:" heading.) Below are four representative examples of the output:

LIST OF BOOKED PASSENGERS:

BEATON/CHRIS

ALLEN/A

BRADY/B

LIST OF CANCELLED PASSENGERS:

ATEST/AL

LIST OF CANCELLED PASSENGERS:

NO CANCELLED ITEMS FOUND

LIST OF BOOKED PASSENGERS:

NO BOOKED ITEMS FOUND

Error Processing:

You should have an error routine that informs the user of instances where the format of the input message is in error. At this point in time, make the routine simple, by using puts() function calls to display some specifics to the user concerning the nature of the format error.

This error routine should also issue an error message if FACS fails, or if your program is unable to find a PNID record. Send a message to the user to keep them informed, and to unlock their keyboard.

Be sure to test all error paths.

Structured Design Chart:

You are required to create a structured design chart before you start coding your solution to this skill check. The chart should show the various modules of the program solution you will be coding, as well as the relationships between the modules as indicated with data connections and control connections. Show your chart to a course manager before you begin coding your solution.

File I/O Skill Check 1 Solution

```
/* This trb1 program receives input from the user indicating     */
/* whether the program is to display booked items in a specified */
/* PNID record, or cancelled items in the specified PNID record. */
/* The PNID record is specified via the PNID ordinal number that */
/* must be provided as input, following a secified B or C input  */
/* to indicate whether booked or cancelled items will be listed. */
/* Input must match one of the following formats:                */
/*                                                               */
/*                t*trb1/b xxxx                                  */
/*                                                               */
/*                    or                                         */
/*                                                               */
/*                t*trb1/c xxxx                                  */
/*                                                               */
/*                                                               */
/* In the above input formats, b and c indicate whether booked   */
/* or cancelled items are being searched for.  xxxx represents   */
/* the PNID record ordinal number.                               */
/*                                                               */
/* This program uses data levels D0 and D4.                      */
/*                                                               */
/*                                                               */


#include "gifirst.h"
#include <tpfeq.h>
#include <tpfio.h>
#include <tpfapi.h>
#include <stdio.h>
#include "gitpf.h"
#ifndef pd1_structure
#define pd1_structure 1

#pragma pack(1);                    /* remove pad bytes */
struct pd1hd              /*header and common area of pd1pd */
{
char pd1bid[2];          /*record id = pd */
char pd1chk;              /*record code check */
char pd1ctl;              /*control byte-bit4 on = chain */
char pd1pgm[4];          /*last program to file this rec */
unsigned long pd1fch;    /*forward chain */
short pd1inx;            /*location index */
char pd1id1;              /*indicator byte */
char pd1id2;              /*another indicator byte */
long pd1flt;              /*flight number, 3=regular,4=extra */
short pd1dat;            /*depart date of last seg - binary */
short pd1sp6;            /*aci indicator byte */
unsigned long pd1wlp;    /*active waitlist file addr */
unsigned long pd1wln;    /*cancelled waitlist file addr */
unsigned long pd1gri;    /*grid file address */
char pd1sp2[16];         /*spares */
};

struct pd1itm              /*structure for pnix item*/
{
unsigned int pd1id3_bit0: 1;        /*cancelled from pni */
unsigned int pd1id3_bit1: 1;        /*group block pnr */
unsigned int pd1id3_bit2: 1;        /*sold from group block pnr */
unsigned int pd1id3_bit3: 1;        /*pta pnr */
unsigned int pd1id3_bit4: 1;        /*name is truncated */
unsigned int pd1id3_bit5: 1;        /*corporate name */
unsigned int pd1id3_bit6: 1;        /*primary - 1st or only name */
unsigned int pd1id3_bit7: 1;        /*facts or supplementary info */
unsigned int pd1id4_all: 8;         /*control byte */
unsigned int pd1sp3_all: 8;         /*another indicator byte */
unsigned int pd1id5_all: 8;          /*indicators-formerly pd1scc */
unsigned char pd1cos;                /*class of service bcd */
```

```
unsigned long pd1add;                /*pnr address */
unsigned char pd1prc;                /*record check code */
char pd1name[12];                    /*passenger name */
char pd1bpt[3];                      /*board point */
char pd1opt[3];                      /*off point */
unsigned char pd1nua;                /*binary# of pass with surnam */
unsigned char pd1npt;                /*binary number in party */
unsigned char pd1bkt;                /*bucket number */
char pd1sp4;                         /*spare */
short pd1bdt;                        /*booking date */
short pd1tdt;                        /*ticket & cancel date */
short pd1kri;                        /*kri item # for sch chg */
char pd1mcc[2];                      /*marketing carrier code */
};

#define RECTYPE "#PNDRI "           /* define PNID record type */
/* define number of items per PNID */
#define NUM_ITM \
( (_LBSZE - sizeof (struct pd1hd) ) / sizeof (struct pd1itm) )
/* _LBSZE is the system #define for a Large Block Size */
struct pd1pd         /* template of PNID record */
{
struct pd1hd d1_header;                 /*header/common area*/
struct pd1itm d1_items[NUM_ITM];       /* items */
char pd1sp5[3];                         /* spares */
};
#endif
/* define LIMIT as number of possible items in PNID record */
#define LIMIT 25


#define BOOKED_HEADER "List of Booked Passengers:"
#define CANCELLED_HEADER "List of Cancelled Passengers:"

void display(char request, struct pd1pd * pnid_ptr);
void errors(short error_num);
void FACS(struct TPF_regs * regs);
void trb1(void)
{
    struct TPF_regs regs;        /* declare registers for FACS */
    struct mi0mi * mi_ptr;       /* ptr to input msg */
    struct pd1pd * pnid_ptr;     /* ptr to PNID record */
    struct pd1itm * item_ptr;    /* ptr to single PNID item */

    char request_type;            /* cancelled (C) or booked (B) */
    char junk;                    /* holds discarded B or C from input */
    long ord;                    /* PNID ord number */
    short num_of_args;           /* scanf return value */
/*****************************************************************/
/* Interrogate input message                                     */
/*****************************************************************/
mi_ptr = (struct mi0mi *) ecbptr()->ce1cr0; /* set ptr to msg blk */
request_type = mi_ptr->mi0acc[7];            /* is input B or C? */

/* The above use of the mi0mi structure is here just to show you
   that you can use the mi0mi structure to access the input message.
   Below, we use scanf() to get other parts of the input message.
   We have already accessed the B or C part of the input, so in the
   the scanf() below, we assign the B or C to a junk variable and
   just discard it.  */
num_of_args = scanf("T*TRB1/ %c %d", &junk, &ord);

if ( request_type != 'B' && request_type != 'C' )   /* invalid type?*/
{
   errors(1);                            /* input type error */
}
```

```
if ( num_of_args < 2 )        /* if fewer than 2 args in input */
{
   errors(1);                                    /* format error */
}

/*****************************************************************/
/* Setup registers for FACS                                      */
/*****************************************************************/
regs.r6 = (long) RECTYPE;
regs.r7 = (long) &(ecbptr()->ce1fa4);
regs.r0 = ord;

/*****************************************************************/
/* Call FACS                                                     */
/*****************************************************************/
FACS(&regs);
if ( regs.r0 == 0)              /* FACS error? */
{
    errors(2);                  /* Yes, FACS error */
}
/*****************************************************************/
/* Retrieve the PNID prime record onto level 4 without hold      */
/* If find fails, call error function.                           */
/*****************************************************************/
pnid_ptr = find_record(D4,NULL,"PD",'\0',NOHOLD)
if ( pnid_ptr == 0 )            /* find error? */
{
    errors(3);                  /* Yes, find error */
}


/*****************************************************************/
/* Call the display function                                     */
/*****************************************************************/
display(request_type, pnid_ptr);
/*****************************************************************/
/* Release resources before program ends and have ECB exit system */
/*****************************************************************/
relcc(D0);            /* release input msg block */
relcc(D4);            /* release PNID core block */
exit(0);              /* release ECB and exit system */
return;
}                     /* end of trb1 function */
/*********************************************************************/
/* This is the display function. It displays either booked or        */
/* cancelled passenger names from the PNID record, according to      */
/* the request type specified in the user input message.             */
/*********************************************************************/
void display(char request,struct pd1pd * pnid_ptr)
{
int index = 0;              /* looping index */
int cancelled = 0;          /* cancelled flag */
int booked = 0;             /* booked flag */
int count;                  /* string count for sprintf */
int actual;                  /* actual number of pd1 items */
char buff[300];             /* buffer for sprintf */
char name_temp[13];         /* temp area for sprintf */
actual = (pnid_ptr->pd1inx) - sizeof(struct pd1hd)/
         sizeof(struct pd1itm)

if ( actual > LIMIT)         /* insure that actual is not too large */
{
     actual = LIMIT;
}

if ( request == 'B' )       /* look for booked items if B input */
```

```
{
     count = sprintf(buff,"%s n",BOOKED_HEADER);

     while ( index < actual )
     {
          if ( pnid_ptr->d1_items[index].pd1id3_bit0 == 0 )
          {
            memcpy(name_temp,pnid_ptr->d1_items[index].pd1name,12);

            name_temp[12] = '\0';        /* add terminating null */

            count+=sprintf(buff+count,"%s\n",name_temp);

            booked = 1;         /* set flag */
          }           /* end of if */
       index++;                    /* bump to next item in record */
     }  /* end of while loop */
     printf("%s",buff);
     if ( booked == 0 ) /* Additional output if no booked items */
     {
         puts("NO BOOKED ITEMS FOUND");
     }   /* end of if */

} /* end of if */

if ( request == 'C' )
{
   count = sprintf(buff,"%s n",CANCELLED_HEADER);
   while ( index < actual )
   {
       if ( (pnid_ptr->d1_items[index].pd1id3_bit0 ))
       {
          memcpy(name_temp,pnid_ptr->d1_items[index].pd1name,12);

          name_temp[12] = '\0';         /* add terminating null */

          count+=sprintf(buff+count,"%s\n",name_temp);

          cancelled = 1;            /* set flag */
       }
       index++;       /* bump to next item in record */

   }  /* end of while loop */
   printf("%s",buff); /* send output */

   if ( cancelled == 0 ) /* Additional output if no cancelled items */
   {
       puts("NO CANCELLED ITEMS FOUND");
   }   /* end of if */

}  /* end of if */
return;
} /* end of display function */
/*****************************************************************/
/* This is the errors function. It displays messages back to the    */
/* user in response to input message format errors, FACS errors,    */
/* or record find errors detected in the program.                   */
/*****************************************************************/
void errors( short error_num)
{
    switch (error_num )
    {
        case 1:     /* input format error */
              puts("The format of the input message is in error");
              break;
```

```
        case 2:     /* FACS failed */
              puts("File address retrieval failed");
              break;


        case 3:     /* find on record failed */
              puts("Record retrieval failed");


    } /* end of switch */

    crusa(2,D0,D4);     /* release core blocks if they exist */

    exit(0);            /* have ECB exit the system */
} /* end of errors function */
```

The TPF C Functions for Writing Files to DASD

The table below describes the primary TPF C functions related to writing TPF records to DASD.
Each C function in the table is a link to a section farther down on this page that describes the function in detail. Click on each function name to jump to that function's corresponding description, and then read that material so that you understand the function's purpose and syntax.

C Function	Assembler macro equivalent	Purpose
file_record()	FILEC FILUC FILNC	Write a record to file, and optionally unhold the record, or optionally do not release the core block
filec()	FILEC	Write to file
filnc()	FILNC	Write to file, but do not release block
filuc()	FILUC	Write to file, and unhold the record
unfrc()	UNFRC	Unhold a record

Below, let's examine each of the above TPF C functions...

file_record() - Write a record to file

Include	tpfio.h
Prototype	void file_record(enum t_lvl ***level***, const unsigned int * ***address***, const char * ***id***, unsigned char ***rcc***, enum t_act ***type***); This function returns nothing.

Parameters	1. ***level*** is a valid data level (D0 - DF). 2. ***address*** is a pointer to the file address of the file to be written. 3. ***id*** is a pointer to the 2-character record ID. 4. ***rcc*** is the 1-character record code check. 5. ***type*** is one of the following: UNHOLD releases exclusivehold by this ECB. NOHOLD indicates the record is not held. NOREL indicates the core block containing the record is not to be released.
Example	`#include "gifirst.">` `#include <tpfio.h>` `#include "gitpf.h"` `struct somefile` `{` `  char file_id[2];` `  char file_rcc;` `  ...` `};` `void pcx1(void)` `{` `  struct somefile * file_ptr;` `  unsigned long addr;` `  char * recid = "SF";` `  char rcc = 23;` `  ...` **`/* Note: This example requires file addrs in CE1FA5 */`** `  file_ptr = find_record(D5,&addr,recid,rcc,NOHOLD);` `  if (file_ptr == 0)` `    {` `      /* Process find error */` `      ...` `    }` `  ...` `  ...` **`  file_record(D5,NULL,NULL,rcc,NOHOLD);`** `  ...`
Usage Notes	1. The file address reference word (FARW) of the specified level is initialized by this function, as well as the core block reference word upon return. 2. If the FARW already contains the correct address or ID, you can code NULL as those parameters. 3. The record ID and RCC parameters are compared to those of the record on file. If the compare fails, system error routines will be invoked. 4. Coding RECID_RESET as the ID parameter will set

	the FARW to binary zeros. 5. To bypass the RCC check, code '\0' as the RCC parameter. 6. If NOREL is coded, a waitc() call should be coded before subsequent use of the unreleased core block. 7. UNHOLD, NOHOLD, and NOREL are mutually exclusive.

filec() - Write a record to file

Include	tpfio.h
Prototype	void filec(enum t_lvl ***level***); This function returns nothing.
Parameter	***level*** is a valid data level (D0 - DF).
Example	`#include "gifirst.">` `#include <tpfio.h>` `#include "gitpf.h"` `...` `...` `void pcx1(void)` `{` `  ...` `  ...` **`/* Note: The FARW must be set up correctly before call to filec() */`** **`  filec(D5);`** `  ...` `  ...`
Usage Notes	1. The file address reference word must contain the correct record ID, record code check and file address. If any of those parameters are incorrect, system error routines will be invoked. 2. The core block on the level specified is released.

filnc() - Write to file with no release of core block

Include	tpfio.h
Prototype	void filnc(enum t_lvl ***level***);

	This function returns nothing.
Parameter	***level*** is a valid data level (D0 - DF).
Example	`#include "gifirst.h">` `#include <tpfio.h>` `#include "gitpf.h"` `...` `...` `void pcx1(void)` `{` `  ...` `  ...` **`/* Note: The FARW must be set up correctly before call to filnc() */`** **`  filnc(D5);`** `  ...` `  ...` **`  /* Block remains attached */`** **`  /* Must issue waitc() to re-use block */`** `  ...`
Usage Notes	1. The file address reference word must contain the correct record ID, RCC and file address. If any of those parameters are incorrect, system error routines will be invoked. 2. The core block on the level specified is accessible by the program, after the waitc() has completed.

filuc() - Write to file and unhold the record

Include	tpfio.h
Prototype	void filuc(enum t_lvl ***level***); This function returns nothing.
Parameter	***level*** is a valid data level (D0 - DF).
Example	`#include "gifirst.h">` `#include <tpfio.h>` `#include "gitpf.h"` `...` `...` `void pcx1(void)` `{` `  ...` `  ...`

	/* Note: The FARW must be set up correctly before call to filuc() */ filuc(D5); ... ...
Usage Notes	1. The file address reference word must contain the correct record ID, RCC and file address. If any of those parameters are incorrect, system error routines will be invoked. 2. The core block on the level specified is released.

unfrc() - Unhold a record

Include	tpfio.h
Prototype	void unfrc(enum t_lvl ***level***); This function returns nothing.
Parameter	***level*** is a valid data level (D0 - DF).
Example	#include "gifirst.h"> #include <tpfio.h> #include "gitpf.h" ... ... void pcx1(void) { ... ... unfrc(D5); ... ...
Usage Notes	1. If the record is not held, system error routines will be invoked. 2. The core block on the level specified is not released.

File Skill Check 2

This exercise builds upon the coding you did for File I/O skill check 1.
We now want to add functionality that accommodates an additional input message. This new input message allows the user to add a passenger name to the PNID record. So,

after retrieving the record, you will be updating the record (adding the name) and filing it out to DASD.
After the name has been placed in the passenger name field of the new item, your program should:

- Set the booked bit in the new PNID item.
- Update the PNID record's location index.
- File and unhold the PNID record.
- Release the core block that held the PNID.

Due to the limited time and purposes of this course, it is not required that your program be able to detect situations where the specified prime PNID does not have space to accommodate a new item. So if you fill up a prime with new items, and you want to make more "add name" entries, then specify a different ordinal number in the input, one that refers to a record that does have space in the prime for additional new items.

Input:

t*trb1/***n ordinal# passengername***

The '***n***' character indicates that the user wishes to add the specified passenger name to the next available slot in the PNID associated with the specified ordinal number.

You should be able to make a subsequent t*trb1/***b ordinal#*** entry (with 'b' specified) to verify that the name has indeed been added to the record, and that the new item is booked.

t*trb1/***b ordinal#***

t*trb1/***c ordinal#***

Your program should retain its ability to process the above booked and cancelled input messages. The details of that processing was addressed in File I/O skill check 1.

Output:

When the input message is of the form

t*trb1/***n ordinal# passengername***

a message should be sent to the user indicating that the name was added to the PNID.

When the input message is such that it requests a list of booked passengers or cancelled passengers, provide output similar that provided in your solution to File I/O skill check 1.

Error Processing:

The error processing requirements are the same as those specified in File I/O skill check 1.

Be sure to test all error paths.

Structured Design Chart:

You are required to create a structured design chart before you start coding your solution to this skill check. The chart should show the various modules of the program solution you will be coding, as well as the relationships between the modules as indicated with data connections and control connections. Show your chart to a course manager before you begin coding your solution.

File I/O Skill Check 2 Solution

```
/*(C) Copyright Galileo International     */
/*Partnership, 1998. All rights rserved. */
/* This trb1 program receives input from the user indicating     */
/* whether the program is to display booked items in a specified */
/* PNID record, or cancelled items in the specified PNID record, */
/* or whether a new name is to be added to the record.           */
/* The PNID record is specified via the PNID ordinal number that */
/* must be provided as input, following a secified B or C input  */
/* to indicate whether booked or cancelled items will be listed, */
/* or an N to indicate that a name is to be added to the PNID.   */
/* Input must match one of the following formats:                */
/*                                                               */
/*                t*trb1/b xxxx                                  */
/*                                                               */
/*                    or                                         */
/*                                                               */
/*                t*trb1/c xxxx                                  */
/*                                                               */
/*                    or                                         */
/*                                                               */
/*                t*trb1/n xxxx ppppppppp                        */
/*                                                               */
/*                                                               */
/* In the above input formats, b and c indicate whether booked   */
/* or cancelled items are being searched for.  xxxx represents   */
/* the PNID record ordinal number. n indicates a new name is to  */
/* is to be added to the prime PNID indicated by the ordinal     */
/* number.  ppppppppp represents the new name.                   */
/*                                                               */
/* This program uses data levels D0 and D4.                      */
/*                                                               */
/*                                                               */
```

```
#include "gifirst.h"
#include <tpfeq.h>
#include <tpfio.h>
#include <tpfapi.h>
#include <stdio.h>
#include "gitpf.h"
#ifndef pd1_structure
#define pd1_structure 1

#pragma pack(1);                    /* remove pad bytes */
struct pd1hd              /*header and common area of pd1pd */
{
char pd1bid[2];          /*record id = pd */
char pd1chk;              /*record code check */
char pd1ctl;              /*control byte-bit4 on = chain */
char pd1pgm[4];          /*last program to file this rec */
unsigned long pd1fch;    /*forward chain */
short pd1inx;            /*location index */
char pd1id1;              /*indicator byte */
char pd1id2;              /*another indicator byte */
long pd1flt;              /*flight number, 3=regular,4=extra */
short pd1dat;            /*depart date of last seg - binary */
short pd1sp6;            /*aci indicator byte */
unsigned long pd1wlp;    /*active waitlist file addr */
unsigned long pd1wln;    /*cancelled waitlist file addr */
unsigned long pd1gri;    /*grid file address */
char pd1sp2[16];         /*spares */
};

struct pd1itm              /*structure for pnix item*/
{
unsigned int pd1id3_bit0: 1;        /*cancelled from pni */
unsigned int pd1id3_bit1: 1;        /*group block pnr */
unsigned int pd1id3_bit2: 1;        /*sold from group block pnr */
unsigned int pd1id3_bit3: 1;        /*pta pnr */
unsigned int pd1id3_bit4: 1;        /*name is truncated */
unsigned int pd1id3_bit5: 1;        /*corporate name */
unsigned int pd1id3_bit6: 1;        /*primary - 1st or only name */
unsigned int pd1id3_bit7: 1;        /*facts or supplementary info */
unsigned int pd1id4_all: 8;         /*control byte */
unsigned int pd1sp3_all: 8;         /*another indicator byte */
unsigned int pd1id5_all: 8;          /*indicators-formerly pd1scc */
unsigned char pd1cos;                /*class of service bcd */
unsigned long pd1add;                /*pnr address */
unsigned char pd1prc;                /*record check code */
char pd1name[12];                    /*passenger name */
char pd1bpt[3];                      /*board point */
char pd1opt[3];                      /*off point */
unsigned char pd1nua;                /*binary# of pass with surnam */
unsigned char pd1npt;                /*binary number in party */
unsigned char pd1bkt;                /*bucket number */
char pd1sp4;                         /*spare */
short pd1bdt;                        /*booking date */
short pd1tdt;                        /*ticket & cancel date */
short pd1kri;                        /*kri item # for sch chg */
char pd1mcc[2];                      /*marketing carrier code */
};

#define RECTYPE "#PNDRI "           /* define PNID record type */
/* define number of items per PNID */
#define NUM_ITM \
( (_LBSZE - sizeof (struct pd1hd) ) / sizeof (struct pd1itm) )
/*  _LBSZE is the system #define for a Large Block Size    */

struct pd1pd         /* template of PNID record */
{
struct pd1hd d1_header;                 /*header/common area*/
```

```
struct pd1itm d1_items[NUM_ITM];       /* items */
char pd1sp5[3];                         /* spares */
};
#endif
/* define LIMIT as number of possible items in PNID record */
#define LIMIT 25


#define BOOKED_HEADER "List of Booked Passengers:"
#define CANCELLED_HEADER "List of Cancelled Passengers:"

void name_add(char * name, struct pd1pd * pnid_ptr);
void display(char request, struct pd1pd * pnid_ptr);
void errors(short error_num);
void FACS(struct TPF_regs * regs);
void trb1(void)
{
    struct TPF_regs regs;        /* declare registers for FACS */
    struct mi0mi * mi_ptr;       /* ptr to input msg */
    struct pd1pd * pnid_ptr;     /* ptr to PNID record */
    struct pd1itm * item_ptr;    /* ptr to single PNID item */

    char new_name[13];             /* for adding a name */
    char request_type;            /* cancelled (C) or booked (B) */
    char junk;                     /* holds discarded B or C from input */
    long ord;                     /* PNID ord number */
    short num_of_args;           /* scanf return value */
/*****************************************************************/
/* Interrogate input message                                     */
/*****************************************************************/
mi_ptr = (struct mi0mi *) ecbptr()->ce1cr0; /* set ptr to msg blk */
request_type = mi_ptr->mi0acc[7];       /* is input B or C or N? */

/* The above use of the mi0mi structure is here just to show you
   that you can use the mi0mi structure to access the input message.
   Below, we use scanf() to get other parts of the input message.
   We have already accessed the B, C or N part of the input, so in the
   the scanf() below, we assign the B, C or N to a junk variable and
   just discard it.  */

if ( request_type != 'B' && request_type != 'C' && request_type != 'N' )
{
   errors(1);             /* input format error */
}

if ( request_type == 'N' )
{
   num_of_args = scanf("T*TRB1/ %c %ld %12s",&junk,&ord,new_name);
   if ( num_of_args < 3 ) /* if input format error */
   {
      errors(1);          /* input format error */
   }
}
else     /* B or C type entry */
   num_of_args = scanf("T*TRB1/ %c %ld", &junk, &ord);
if ( num_of_args < 2 )
{
   errors(1);             /* if input format error */
}

/*****************************************************************/
/* Setup registers for FACS                                      */
/*****************************************************************/
regs.r6 = (long) RECTYPE;
regs.r7 = (long) &(ecbptr()->ce1fa4);
regs.r0 = ord;
```

```
/*****************************************************************/
/* Call FACS                                                     */
/*****************************************************************/
FACS(&regs);
if ( regs.r0 == 0)              /* FACS error? */
{
    errors(2);                  /* Yes, FACS error */
}
/*****************************************************************/
/* Retrieve the PNID prime record onto level 4.  HOLD or NOHOLD  */
/*depends on request type provided by user.                      */
/* If find fails, error function called.                         */
/*****************************************************************/
if ( request_type == 'N' )      /* new name to be added */
{
   pnid_ptr = find_record(D4,NULL,"PD",'\0',HOLD)
}

else                             /* B or C input */
   pnid_ptr = find_record(D4,NULL,"PD",'\0',NOHOLD)
if ( pnid_ptr == 0 )            /* find error? */
{
   errors(3);                  /* Yes, find error */
}


/*****************************************************************/
/* Call the display or name_add function, depending on input     */
/*****************************************************************/
switch ( request_type )
{
  case 'B':
           display(request_type, pnid_ptr);
           break;
  case 'C':
           display(request_type, pnid_ptr);
           break;
  case 'N':
           name_add(new_name, pnid_ptr);
           break;
}                /* end of switch */

/*****************************************************************/
/* Release resources before program ends and have ECB exit system */
/*****************************************************************/
crusa(2,D0,D4);       /* release input block and PNID block */
exit(0);              /* release ECB and exit system */
return;
}                     /* end of trb1 function */
/*****************************************************************/
/* This is the display function. It displays either booked or    */
/* cancelled passenger names from the PNID record, according to  */
/* the request type specified in the user input message.         */
/*****************************************************************/
void display(char request,struct pd1pd * pnid_ptr)
{
int index = 0;              /* looping index */
int cancelled = 0;          /* cancelled flag */
int booked = 0;             /* booked flag */
int count;                  /* string count for sprintf */
int actual;                  /* actual number of pd1 items */
char buff[300];             /* buffer for sprintf */
char name_temp[13];         /* temp area for sprintf */
actual = (pnid_ptr->pd1inx) - sizeof(struct pd1hd)/
         sizeof(struct pd1itm);
if (actual > LIMIT)          /* insure actual is not too big */
{
```

```
     actual = LIMIT;
}
if ( request == 'B' )       /* look for booked items if B input */
{
     count = sprintf(buff,"%s n",BOOKED_HEADER);
     while ( index < actual )
     {
          if ( pnid_ptr->d1_items[index].pd1id3_bit0 == 0 )
          {
            memcpy(name_temp,pnid_ptr->d1_items[index].pd1name,12);

            name_temp[12] = '\0';        /* add terminating null */

            count+=sprintf(buff+count,"%s\n",name_temp);

            booked = 1;          /* set flag */
          }           /* end of if */
       index++;                    /* bump to next item in record */
     }  /* end of while loop */
     printf("%s",buff);
     if ( booked == 0 ) /* Additional output if no booked items */
     {
         puts("NO BOOKED ITEMS FOUND");
     }   /* end of if */

} /* end of if */

if ( request == 'C' )
{
   count = sprintf(buff,"%s n",CANCELLED_HEADER);
   while ( index < actual )
   {
       if ( (pnid_ptr->d1_items[index].pd1id3_bit0 ))
       {
          memcpy(name_temp,pnid_ptr->d1_items[index].pd1name,12);

          name_temp[12] = '\0';         /* add terminating null */

          count+=sprintf(buff+count,"%s\n",name_temp);

          cancelled = 1;            /* set flag */
       }
       index++;       /* bump to next item in record */

   }  /* end of while loop */
   printf("%s",buff); /* send output */

   if ( cancelled == 0 ) /* Additional output if no cancelled items */
   {
       puts("NO CANCELLED ITEMS FOUND");
   }   /* end of if */

}  /* end of if */
return;
} /* end of display function */


/*********************************************************************/
/* This is the name_add function. It places a passenger name         */
/* (provided as user input) in the next available item in the PNID.  */
/* The booked indicator is turned on, the location index is updated, */
/* and the PNID record is filed and unheld.                          */
/*********************************************************************/
void name_add( char * name, struct pd1pd * pnid_ptr )
{
  short length;                    /* For number of name letters */
  short k;                         /* index into PNID item array */
```

```
  k = (pnid_ptr->d1_header.pd1inx - sizeof(struct pd1hd))/
                  sizeof(struct pd1itm);
  length = strlen(name);

  while( length < 12 )              /* replace terminating null and */
  {                         /* and remaining array elements with spaces */
    name[length] = ' ';
    length++;
  }
  /* place name in PNID */
  memcpy(pnid_ptr->d1_items[k].pd1name, name, 12);

  pnid_ptr->d1_items[k].pd1id3_bit0 = 0;     /* set as booked */

  /* update location index */
  pnid_ptr->d1_header.pd1inx += sizeof(struct pd1itm);

  filuc(D4);            /* file and unhold prime PNID */

  puts("Name added to PNID");
  return;
}      /* end of name_add function */
/*********************************************************************/
/* This is the errors function. It displays messages back to the     */
/* user in response to input message format errors, FACS errors,     */
/* or record find errors detected in the program.                    */
/*********************************************************************/
void errors( short error_num)
{
    switch (error_num )
    {
        case 1:     /* input format error */
              puts("The format of the input message is in error");
              break;


        case 2:     /* FACS failed */
              puts("File address retrieval failed");
              break;


        case 3:     /* find on record failed */
              puts("Record retrieval failed");


    } /* end of switch */

    crusa(2,D0,D4);    /* release core blocks if they exist */

    exit(0);           /* have ECB exit the system */
} /* end of errors function */
```

The TPF C Functions for Managing Pool Files

The table below describes the primary TPF C functions related to obtaining and releasing TPF pool file addresses.
Each C function in the table is a link to a section farther down on this page that describes the function in detail. Click on each function name to jump to that function's corresponding description, and then read that material so that you understand the function's purpose and syntax.

C Function	Assembler macro equivalent	Purpose

getfc()	GETFC	Get file pool address and optionally a working storage block
relfc()	RELFC	Release a pool file address
rlcha()	RLCHA	Release a chain of pool file addresses

Below, let's examine each of the above TPF C functions...

getfc() - Get a pool file address and possibly a working storage block

Include	tpfio.h
Prototype	unsigned int getfc(enum t_lvl ***level***, int ***type***, const char * ***id***, int ***block***, int ***error***, ***...***); This function returns an unsigned integer containing the file address issued by the system.
Parms	1. ***level*** is a valid data level (D0 - DF). 2. ***type*** must specify at least one of the following to determine the pool file attributes: **GETFC_TYPE0 -> TYPE9, GETFC PRIME or GETFC_OVERFLOW** Use one of these options to specify a 2 character record ID from the RIAT table. Prime and Overflow options are supported for migration purposes only and relate to **TYPE0** and **TYPE1** respectively. 3. ***id*** is a pointer to a 2 character record id that is used to scan the RIAT table for the record's attributes. 4. ***block*** must specify one of the following: **GETFC_BLOCK** to obtain a working storage block. **GETFC_NOBLOCK** for no working storage block. In addition to the above options, the block can include optional parameters, each of which must be separated by the logical "or" character (" | "). The optional block parameters are the following: **GETFC_FILL** to initialize the working storage block. **GETFC_COMM** to get a common block. This will cause a dump if no runtime authorization has been obtained. 5. ***error*** must specify one of the following: **GETFC_SERRC** to transfer control to the system error routine if a file address or core storage could not be obtained. Will SERCC with exit.

	GETFC_NOSERRC to return control to the program if there is an error. Return value will be 0 if there is an error. 6. ***"..."*** If **GETFC_FILL** is coded, this parameter will contain the character that is to be used. It may be specified as a single character in single quotes, or as a hex value.
Example	`#include "gifirst.h">` `#include <tpfio.h>` `#include "gitpf.h"` `void pcx1(void)` `{` `  unsigned long blk1, blk2, blk3;` `  ...` `blk1 = getfc(D3,GETFC_TYPE0,"PD",GETFC_BLOCK,GETFC_SERRC);` `...` `...` `blk2 = getfc(D4,GETFC_TYPE1,"PD",GETFC_BLOCK|GETFC_FILL,GETFC_SERRC,' ');` `...` `...`
Usage Notes	1. The specified data level specified must not be holding a block if the **GETFC_BLOCK** parameter is coded. 2. Runtime authorization is required to use the **GETFC_COMM** parameter. 3. If the **GETFC_NOSERRC** parameter is coded and an error occurs, the return value will be zero. 4. If the **GETFC_BLOCK** parameter is coded, the core block reference word for the specified level will contain the address of the core block. 5. The use of this function may result in the equivalent of a waitc(). 6. This function sets the file address reference word to the obtained pool file address automatically.

relfc() - Release a pool file address

Include	tpfio.h
Prototype	void relfc(enum t_lvl ***level***); This function returns nothing. The pool file address in the specified file address reference word (FARW) will be released to the system for reuse.
Parameter	***level*** is a valid data level (D0 - DF).

Example	`#include "gifirst.h">` `#include <tpfio.h>` `#include <tpfeq.h>` `#include "gitpf.h"` `void pcx1(void)` `{` `  struct pd1pd *pd1ptr;` `  ...` `  ...` **`/* copy forward chain address into FARW */`** `  ecbptr()->ebcfa7 = pd1ptr->pd1fch;` **`/* release address of chain */`** **`  relfc(D7);`** `  ...` `  ...`
Usage Notes	1. A system error will occur if the specified FARW does not contain a valid file address. 2. The foward chain field of the record previously associated with the released address should be cleared after the release.

rlcha() - Release a chain of file address

Include	tpfapi.h
Prototype	void rlcha(struct stdhdr * ***hdr***); This function returns nothing.
Parameter	***hdr*** is a pointer to structure that describes the layout of the standard TPF record header.
Example	`#include "gifirst.h"` `#include <tpfapi.h>` `#include "gitpf.h"` `void pcx1(void)` `{` `  struct stdhdr * hdrptr;` `  ...` `  /* Get Prime onto level 6 */` `  ...` `  hdrptr = ecbptr()->ebccr6;`

	`rlcha(hdrptr);`
Usage Notes	1. This function releases pool file addresses from the first record specified to the end of the chain, using the standard TPF record header. The input parameter ***hdr*** may point to any standard header, but the record ID and record code check (RCC) for all records in the chain must match the input parameter header layout. 2. If succeeding records do not have the same ID and RCC, the RLCH subroutine exits.

File Skill Check 3

This exercise builds upon the coding required for File I/O skill check 2. The functionality we created for File I/O skill checks 1 and 2 is still needed, and in this exercise you will gain experience with obtaining pool addresses for the purpose of creating a chain record.
Due to the limited time and purposes of this course, initialize the chain in the following limited manner:

- Copy the header of the prime into the header of the chain.
- Set the location index in the chain header with the proper value.
- Update the prime's forward chain field with the file address of the chain.

As in File I/O skill check 2, it is not required that your program be able to detect situations where the specified PNID does not have space to accommodate a new item when adding a name. If you fill up a prime with new items, and you want to make more "add name" entries, then specify a different ordinal number in the input, one that refers to a record that does have space in the prime for additional new items.

Input:

This new functionality will accommodate the following additional input message:

t*trb1/***a ordinal#**

The '***a***' character indicates that the user wishes to chain a pool file record to the prime PNID associated with the specified ordinal number.

It should also process the entries you coded for in File I/O skill checks 1 and 2, which are the following:

t*trb1/***b ordinal#** (list booked)

t*trb1/***c ordinal#*** (list cancelled***)***

t*trb1/***n ordinal# passengername*** (add name***)***

Output:

If your program adds a chain, it should send a message to the user indicating that a chain was added. If the prime already has a chain, then display a message to the user indicating such, and do not add a chain.

Your program should still provide appropriate output for the three input messages associated with the previous exercise, which was File I/O skill check 2.

Error Processing:

The error processing requirements are the same as those specified in File I/O skill check 2.

Be sure to test all error paths.

Structured Design Chart:

You are required to create a structured design chart before you start coding your solution to this skill check. The chart should show the various modules of the program solution you will be coding, as well as the relationships between the modules as indicated with data connections and control connections. Show your chart to a course manager before you begin coding your solution.

Functionality Test for File I/O Skill Check 3

The following is a functionality check sheet for your **File I/O skill check 3 solution.** You are to print this form, and then ask another student in the class to observe the running of your program, as you demonstrate to that student that your program does provide the functionality describe on this check sheet. When finished, make sure the information below has been filled in, and then present this check sheet to a course manager for skill check sign-off.

Programmer Name :_____

Person who observered the functionality :_____
Observer comments: _____
_____
_____
Observer sign-off signature: _____

Listing Booked Passengers

- Check to see that the **t*trb1/b *ordinal*** entry results in the display of a list of booked passengers when there are booked passengers in the PNID.

Listing Cancelled Passengers

- Check to see that the t*trb1/c ***ordinal*** entry results in the display of a list of cancelled passengers when there are cancelled passengers in the PNID.

Adding a Passenger Name

- Check to see that the **t*trb1/n *ordinal* passengername** entry results in the adding of a booked passenger to the PNID. Follow this entry with the booked list entry to check to see that indeed the name you added now appears when you make the booked list entry.

Adding a Chain to the Prime

- Check to see that the **t*trb1/a *ordinal*** entry results in the adding of a chain to the prime PNID. Make this entry a second time to check for a response that indicates that the prime already has a chain.

Error Processing

- Your program should provide basic input message format error responses to the screen. For example, an appropriate response should be displayed if the letter after the slash is not an ‘b’, ‘c’, ‘n’, or an ‘a’.
- With the add name entry, an error response should appear when a name is not provided in the input.
- An error response should appear anytime an ordinal number is not provided.

File I/O Skill Check 3 Solution

```
/*(C) Copyright Galileo International     */
/*Partnership, 1998. All rights rserved. */
/* This trb1 program receives input from the user indicating     */
/* whether the program is to display booked items in a specified */
/* PNID record, or cancelled items in the specified PNID record, */
/* whether a new name is to be added to the record, or if a new  */
/* chain is to be added to a PNID prime record.                   */
/* The PNID record is specified via the PNID ordinal number that */
/* must be provided as input, following a secified B or C input  */
/* to indicate whether booked or cancelled items will be listed, */
/* or an N to indicate that a name is to be added to the PNID,   */
/* or an A to indicate that a chain is to be added to the prime. */
/* Input must match one of the following formats:                */
/*                                                               */
/*                t*trb1/b xxxx                                  */
/*                                                               */
/*                    or                                         */
/*                                                               */
/*                t*trb1/c xxxx                                  */
/*                                                               */
/*                    or                                         */
/*                                                               */
```

```
/*                t*trb1/n xxxx ppppppppp                          */
/*                                                                 */
/*                    or                                           */
/*                                                                 */
/*                t*trb1/a xxxx                                    */
/*                                                                 */
/*                                                                 */
/* In the above input formats, b and c indicate whether booked     */
/* or cancelled items are being searched for.  xxxx represents     */
/* the PNID record ordinal number. n indicates a new name is to    */
/* is to be added to the prime PNID indicated by the ordinal       */
/* number.  ppppppppp represents the new name.                     */
/* The letter 'a' means add a chain to the prime if a chain        */
/* doesn't already exist.                                           */
/*                                                                 */
/* This program uses data levels D0, D4 and D5                     */
/*                                                                 */
/*                                                                 */

#include "gifirst.h"
#include <tpfeq.h>
#include <tpfio.h>
#include <tpfapi.h>
#include <stdio.h>
#include "gitpf.h"
#ifndef pd1_structure
#define pd1_structure 1

#pragma pack(1);                    /* remove pad bytes */
struct pd1hd              /*header and common area of pd1pd */
{
char pd1bid[2];          /*record id = pd */
char pd1chk;              /*record code check */
char pd1ctl;              /*control byte-bit4 on = chain */
char pd1pgm[4];          /*last program to file this rec */
unsigned long pd1fch;    /*forward chain */
short pd1inx;            /*location index */
char pd1id1;              /*indicator byte */
char pd1id2;              /*another indicator byte */
long pd1flt;              /*flight number, 3=regular,4=extra */
short pd1dat;            /*depart date of last seg - binary */
short pd1sp6;            /*aci indicator byte */
unsigned long pd1wlp;    /*active waitlist file addr */
unsigned long pd1wln;    /*cancelled waitlist file addr */
unsigned long pd1gri;    /*grid file address */
char pd1sp2[16];         /*spares */
};

struct pd1itm              /*structure for pnix item*/
{
unsigned int pd1id3_bit0: 1;        /*cancelled from pni */
unsigned int pd1id3_bit1: 1;        /*group block pnr */
unsigned int pd1id3_bit2: 1;        /*sold from group block pnr */
unsigned int pd1id3_bit3: 1;        /*pta pnr */
unsigned int pd1id3_bit4: 1;        /*name is truncated */
unsigned int pd1id3_bit5: 1;        /*corporate name */
unsigned int pd1id3_bit6: 1;        /*primary - 1st or only name */
unsigned int pd1id3_bit7: 1;        /*facts or supplementary info */
unsigned int pd1id4_all: 8;         /*control byte */
unsigned int pd1sp3_all: 8;         /*another indicator byte */
unsigned int pd1id5_all: 8;          /*indicators-formerly pd1scc */
unsigned char pd1cos;                /*class of service bcd */
unsigned long pd1add;                /*pnr address */
unsigned char pd1prc;                /*record check code */
char pd1name[12];                    /*passenger name */
char pd1bpt[3];                      /*board point */
char pd1opt[3];                      /*off point */
```

```
unsigned char pd1nua;                /*binary# of pass with surnam */
unsigned char pd1npt;                /*binary number in party */
unsigned char pd1bkt;                /*bucket number */
char pd1sp4;                         /*spare */
short pd1bdt;                        /*booking date */
short pd1tdt;                        /*ticket & cancel date */
short pd1kri;                        /*kri item # for sch chg */
char pd1mcc[2];                      /*marketing carrier code */
};

#define RECTYPE "#PNDRI "           /* define PNID record type */
/* define number of items per PNID */
#define NUM_ITM \
( (_LBSZE - sizeof (struct pd1hd) ) / sizeof (struct pd1itm) )
/*  _LBSZE is the system #define for a Large Block Size     */

struct pd1pd         /* template of PNID record */
{
struct pd1hd d1_header;                 /*header/common area*/
struct pd1itm d1_items[NUM_ITM];       /* items */
char pd1sp5[3];                         /* spares */
};
#endif
/* define LIMIT as number of possible items in PNID record */
#define LIMIT 25


#define BOOKED_HEADER "List of Booked Passengers:"
#define CANCELLED_HEADER "List of Cancelled Passengers:"

void add_chain(struct pd1pd * pnid_ptr);
void name_add(char * name, struct pd1pd * pnid_ptr);
void display(char request, struct pd1pd * pnid_ptr);
void errors(short error_num);
void FACS(struct TPF_regs * regs);
void trb1(void)
{
    struct TPF_regs regs;        /* declare registers for FACS */
    struct mi0mi * mi_ptr;       /* ptr to input msg */
    struct pd1pd * pnid_ptr;     /* ptr to PNID record */
    struct pd1itm * item_ptr;    /* ptr to single PNID item */

    char new_name[13];             /* for adding a name */
    char request_type;            /* cancelled (C) or booked (B) */
    char junk;                    /* holds discarded B or C from input */
    long ord;                    /* PNID ord number */
    short num_of_args;           /* scanf return value */
/*****************************************************************/
/* Interrogate input message                                     */
/*****************************************************************/
mi_ptr = (struct mi0mi *) ecbptr()->ce1cr0; /* set ptr to msg blk */
request_type = mi_ptr->mi0acc[7];    /* is input B or C or N or A? */

/* The above use of the mi0mi structure is here just to show you
   that you can use the mi0mi structure to access the input message.
   Below, we use scanf() to get other parts of the input message.
   We have already accessed the B, C, N or A part of the input, so in
   the scanf() below, we assign the B, C, N or A to a junk variable
   and just discard it.  */

if ( request_type != 'B' && request_type != 'C' &&
         request_type != 'N' && request_type != 'A' )
{
   errors(1);             /* input format error */
}

if ( request_type == 'N' )
```

```
{
   num_of_args = scanf("T*TRB1/ %c %ld %12s",&junk,&ord,new_name);

   if ( num_of_args < 3 ) /* if input format error */
   {
      errors(1);          /* input format error */
   }
}
else     /* B C or A type entry */
   num_of_args = scanf("T*TRB1/ %c %ld", &junk, &ord);
if ( num_of_args < 2 )
{
   errors(1);             /* if input format error */
}

/*****************************************************************/
/* Setup registers for FACS                                      */
/*****************************************************************/
regs.r6 = (long) RECTYPE;
regs.r7 = (long) &(ecbptr()->ce1fa4);
regs.r0 = ord;

/*****************************************************************/
/* Call FACS                                                     */
/*****************************************************************/
FACS(&regs);
if ( regs.r0 == 0)               /* FACS error? */
{
    errors(2);                   /* Yes, FACS error */
}
/*****************************************************************/
/* Retrieve the PNID prime record onto level 4.  HOLD or NOHOLD  */
/*depends on request type provided by user.                      */
/* If find fails, error function called.                         */
/*****************************************************************/
if ( request_type == 'N' || request_type == 'A' )
{
   pnid_ptr = find_record(D4,NULL,"PD",'\0',HOLD)
}

else                              /* B or C input */
   pnid_ptr = find_record(D4,NULL,"PD",'\0',NOHOLD)
if ( pnid_ptr == 0 )             /* find error? */
{
   errors(3);                   /* Yes, find error */
}


/*****************************************************************/
/* Call the display, name_add, or add_chain function,            */
/* depending upon input request.                                 */
/*****************************************************************/
switch ( request_type )
{
  case 'A':
           add_chain(pnid_ptr);
           break;
  case 'B':
           display(request_type, pnid_ptr);
           break;
  case 'C':
           display(request_type, pnid_ptr);
           break;
  case 'N':
           name_add(new_name, pnid_ptr);
           break;
}                /* end of switch */
```

```
/*****************************************************************/
/* Release resources before program ends and have ECB exit system */
/*****************************************************************/
crusa(2,D0,D4);       /* release input block and PNID block */
exit(0);              /* release ECB and exit system */
return;
}                     /* end of trb1 function */
/*********************************************************************/
/* This is the add_chain function. It gets a pool record, sets up     */
/* the PNID header on the new chain, puts the new chain's file addrs  */
/* in the forward chain field of the PNID prime, files the new chain, */
/* files and unholds the updated prime.  Level D5 is used for chain.  */
*//*********************************************************************/
void add_chain(_Packed struct pd1pd * pnid_ptr)
{
  struct pd1pd * chain_ptr;
  unsigned long faddr;

  if ( pnid_ptr->d1_header.pd1fch == 0 )    /* if no exiting chain */
  {
    faddr = getfc(D5,GETFC_TYPE1,"PD",GETFC_BLOCK,GETFC_SERRC);
    chain_ptr = ecbptr()->ce1cr5;
    chain_ptr->d1_header = pnid_ptr->d1_header;
    chain_ptr->d1_header.pd1inx = sizeof(struct pd1hd);
    pnid_ptr->d1_header.pd1fch = faddr;
    memcpy(&ecbptr()->ebcid5,"PD",2);     /* put id in FARW */
    filec(D5);            /* file the new chain */
    filuc(D4);          /* file and unhold prime */
    puts("Chain added to prime");
  }
  else     /* prime already has chain */
  {
    puts("Chain already exists");
    unfrc(D4);     /* unhold the prime */
    relcc(D4);     /* release the prime block */
  }
return;
}     /* end of add_chain function */
/*********************************************************************/
/* This is the display function. It displays either booked or        */
/* cancelled passenger names from the PNID record, according to      */
/* the request type specified in the user input message.             */
/*********************************************************************/
void display(char request,struct pd1pd * pnid_ptr)
{
int index = 0;              /* looping index */
int cancelled = 0;          /* cancelled flag */
int booked = 0;             /* booked flag */
int count;                  /* string count for sprintf */
int actual;                  /* actual number of pd1 items */
char buff[300];             /* buffer for sprintf */
char name_temp[13];         /* temp area for sprintf */
actual = (pnid_ptr->pd1inx) - sizeof(struct pd1hd)/
         sizeof(struct pd1itm)
if (actual > LIMIT)          /* prevent actual from being too big */
{
     actual = LIMIT;
}
if ( request == 'B' )       /* look for booked items if B input */
{
     count = sprintf(buff,"%s n",BOOKED_HEADER);
     while ( index < actual )
     {
          if ( pnid_ptr->d1_items[index].pd1id3_bit0 == 0 )
          {
            memcpy(name_temp,pnid_ptr->d1_items[index].pd1name,12);
```

```
            name_temp[12] = '\0';        /* add terminating null */

            count+=sprintf(buff+count,"%s\n",name_temp);

            booked = 1;          /* set flag */
          }           /* end of if */
        index++;                    /* bump to next item in record */
      }  /* end of while loop */
      printf("%s",buff);
      if ( booked == 0 ) /* Additional output if no booked items */
      {
          puts("NO BOOKED ITEMS FOUND");
      }   /* end of if */

} /* end of if */

if ( request == 'C' )
{
   count = sprintf(buff,"%s n",CANCELLED_HEADER);
   while ( index < actual )
   {
       if ( (pnid_ptr->d1_items[index].pd1id3_bit0 ))
       {
          memcpy(name_temp,pnid_ptr->d1_items[index].pd1name,12);

          name_temp[12] = '\0';         /* add terminating null */

          count+=sprintf(buff+count,"%s\n",name_temp);

          cancelled = 1;            /* set flag */
       }
       index++;       /* bump to next item in record */

   }  /* end of while loop */
   printf("%s",buff); /* send output */

   if ( cancelled == 0 ) /* Additional output if no cancelled items */
   {
       puts("NO CANCELLED ITEMS FOUND");
   }   /* end of if */

}  /* end of if */
return;
} /* end of display function */


/*********************************************************************/
/* This is the name_add function. It places a passenger name         */
/* (provided as user input) in the next available item in the PNID.  */
/* The booked indicator is turned on, the location index is updated, */
/* and the PNID record is filed and unheld.                          */
/*********************************************************************/
void name_add( char * name, struct pd1pd * pnid_ptr )
{
  short length;                    /* For number of name letters */
  short k;                         /* index into PNID item array */
  k = (pnid_ptr->d1_header.pd1inx - sizeof(struct pd1hd))/
                 sizeof(struct pd1itm);
  length = strlen(name);

  while( length < 12 )             /* replace terminating null and */
  {                      /* and remaining array elements with spaces */
    name[length] = ' ';
    length++;
  }
```

```
  /* place name in PNID */
  memcpy(pnid_ptr->d1_items[k].pd1name, name, 12);

  pnid_ptr->d1_items[k].pd1id3_bit0 = 0;     /* set as booked */

  /* update location index */
  pnid_ptr->d1_header.pd1inx += sizeof(struct pd1itm);

  filuc(D4);            /* file and unhold prime PNID */

  puts("Name added to PNID");
  return;
}      /* end of name_add function */
/*********************************************************************/
/* This is the errors function. It displays messages back to the     */
/* user in response to input message format errors, FACS errors,     */
/* or record find errors detected in the program.                    */
/*********************************************************************/
void errors( short error_num)
{
    switch (error_num )
    {
        case 1:     /* input format error */
              puts("The format of the input message is in error");
              break;


        case 2:     /* FACS failed */
              puts("File address retrieval failed");
              break;


        case 3:     /* find on record failed */
              puts("Record retrieval failed");


    } /* end of switch */

    crusa(2,D0,D4);    /* release core blocks if they exist */

    exit(0);           /* have ECB exit the system */
} /* end of errors function */
```

CHAPTER 14

This chapter concerns the TPF C functions dealing with error processing. TPF C adds extra functionality over the standard TPF macro set with the inclusion of the abort and assert functions.

Proceed through the chapter by reading all sections in order, beginning with the chapter objectives.

Objectives

The purpose of this chapter is to familiarize you with various functions that exist for the purpose of processing errors.
By the end of the session the student will, with the aid of course material, be able to code relevant error handling functions, such as the following:

- **abort()**
- **assert()**
- **serrc_op()**
- **snapc()**

TPF C Error Functions

The table below describes the primary TPF C functions related to error processing. Each C function in the table is a link to a section farther down on this page that describes the function in detail. Click on each function name to jump to that function's corresponding description, and then read that material so that you understand the function's purpose and syntax.

C Function	Assembler macro equivalent	Purpose
abort()	N/A	Abnormal termination of ECB
assert()	N/A	Print diagnostic message and exit
serrc_op()	SERRC	Issue System Error
snapc()	SNAPC	Issue Snapshot Dump

Below, let's examine each of the above TPF C functions...

abort() - Terminate a process abnormally

Include	stdlib.h
Prototype	void abort(void); The function returns nothing, and the program terminates.
Parameters	None

Example	`#include <tpfeq.h>` `short int count = 0;` `. . .` `. . .` `if(count == 100)` `{` **`abort();`** `}` `. . .` `. . .`
Usage Notes	1. abort() causes dumps to be suppressed. For example, when the ECB is exiting the system due to an abort(), a dump due to the fact that the ECB is holding a record would not occur. 2. This function can be used for exiting an ECB in abnormal circumstances when a serrc is not appropriate. 3. No return is made to the calling program.

assert() - Print a diagnostic message

Include	assert.h
Prototype	void assert(int ***expression***); The function returns nothing, and the program terminates.
Parameter	***expression*** is a valid TPF C expression that is to be tested.
Example	`#include <assert.h>` `. . .` `short int count = 0` `. . .` `. . .` **`assert(count < 100);`** `. . .` `. . .`
Usage Notes	1. If ***expression*** is true, this function has no effect. If the expression is false, a message is displayed on the agent set and the ECB is exited using the abort() function. Message format and example:

	Assertion failed: <expression>, file: <program name>, line: <line#> Assertion failed: count < 100, file: PCX1 C A, line: 27. 2. This function should only be used for the testing phase of a project as a diagnostic tool. To nullify the assert, sometime before loading the code, you should add the line **#define NDEBUG 1**. When the NDEBUG 1 and **#include <assert.h>** are found in the same Source Component by the compiler, all asserts in that Source Component are changed to null expressions (equivalent to NOOP instructions in BAL). 3. No return is made to the calling program.

serrc_op() - Issue System Error

Include	tpfapi.h
Prototype	void serrc_op(enum t_serrc ***status***, int ***number***, const char * ***msg***, void * ***slist[ ]***); The function returns nothing.
Parameters	1. ***status*** indicates the status of the ECB following the serrc. Code **SERRC_EXIT** to exit the ECB or **SERRC_RETURN** to return to the program issuing the dump. 2. ***number*** is a unique integer value ranging from x'1' through x'FFFFFF'. Will be prefixed with letter U in the dump. To change the prefix, use the serrc_op_ext() version of this function. 3. ***msg*** is a pointer to the message to be appended to the SERRC message on the CRAS set. If no message is required, code NULL. 4. ***slist[ ]*** is a pointer to an array of pointers to additional areas of storage that are to be dumped. If no storage list exists, code NULL.
Example	`#include <tpfapi.h>` `...` `...` `if(error_condition == 1)` `{` **`serrc_op(SERRC_EXIT,0x123321,"OOPS",NULL);`** `}` `...`

	. . .
Usage Notes	1. The function Issues a **SERCC** with exit or return, depending of the specified parameter. 2. Maximum length of the message is 255 characters, which must end with a terminating null. 3. The message generated goes to the Prime CRAS console, not to the user's terminal. If a message is to be sent to the user's terminal, you need to code a return and then issue the user's message. 4. If the message coded is greater than 255 characters, the message will be truncated. 5. See the description of the snapc() function for information on using the slist[ ] parameter.

snapc() - Issue Snapshot Dump

Include	tpfapi.h
Prototype	void snapc(int ***action***, int ***code***, const char * ***msg***, struct snapc_list ** ***listc***, char ***prefix***, int ***regs***, int ***ecb***, const char * ***program***); The function returns nothing.
Parameters	1. ***action*** indicates whether, after the dump, the ECB exits or processing continues. Code **SNAPC_EXIT** to exit after the dump; code **SNAPC_RETURN** to return and continue processing after the dump. 2. ***code*** is a unique dump number ranging from x'0' through x'7FFFFFFF'. 3. ***msg*** is a message to be appended to the snapc message that goes to the CRAS set and appears in the dump. If no message is required, specify NULL. 4. ***listc*** is a pointer to an array of pointers that point to extra areas of storage that are to be dumped. If you do not wish to dump such extra areas, then specify NULL for this parameter. The following are the four members of the snapc_list structure... **snapc_len** is a short int, and is used to indicate the size of area to dump. Specify zero to indicate end of list. **snapc_name** is to contain the character string name of the area to be dumped. The name must be 8 bytes,

	left justified. **snapc_tag** is to contain the address of area to be dumped. **snapc_indir** is set to either **SNAPC_INDIR** or **SNAPC_NOINDIR**, to indicate whether the snapc_tag member points to an address or to data. 5. ***prefix*** is a single character that is to serve as the dump number. You can use A through H, or J through V. The remaining characters are reserved for IBM. 6. ***regs*** is set to **SNAPC_REGS** to include the registers in the dump, or set to **SNAPC_NOREGS** so that registers are not included. 7. ***ecb*** is set to **SNAPC_ECB** to get SSU name, SS name and the terminal address from the ECB. Set it to **SNAPC_NOECB** to get BSS names. No terminal address will be available if **SNAPC_NOECB** is specified. 8. ***program*** is a character string containing the name of the program. Maximum of 16 characters. Specify NULL if you want the name to be the current program's name.
Example	The following is a snapc() example...

```
. . .
. . .

void errors(long ordnum, char rtype[])
{

   struct snapc_list * cvsnap[3], slist[3];


   /* set up addressability to snap list */
   cvsnap[0] = &slist [0];
   cvsnap[1] = &slist [1];
   cvsnap[2] = &slist [2];

   /* initialize snap list for record type */
   cvsnap[0]->snapc_len = strlen (rectype);
   cvsnap[0]->snapc_name = "REC TYPE";
   cvsnap[0]->snapc_tag = rtype;
   cvsnap[0]->snapc_indir = SNAPC_NOINDIR;

   /* initialize snap list to dump ord number */
   cvsnap[1]->snapc_len = sizeof (ordnum);
   cvsnap[1]->snapc_name = "ORD NUM ";
   cvsnap[1]->snapc_tag = &ordnum;
   cvsnap[1]->snapc_indir = SNAPC_NOINDIR;
```

```
   /* zero length indicates end of the list */
   cvsnap[2]->snapc_len = 0;

   /* call the snapc() function */
   snapc(SNAPC_EXIT, 0x11111111,
      "FACS ERR: REC TYPE, ORD NO, & REGISTERS",
      cvsnap, 'U', SNAPC_REGS, SNAPC_ECB, NULL);

   return;
}
```

Usage Notes	1. The function Issues a **SNAPC** with exit or return, depending of the specified parameter. 2. Maximum length of the message is 255 characters, which must end with a terminating null. 3. The message generated goes to the Prime CRAS console, not to the user's terminal. If a message is to be sent to the user's terminal, you need to code a return and then issue the user's message. 4. If the message coded is greater than 255 characters, the message will be truncated.

Error Processing Skill Check

Add error functions to your File I/O skill check 3 solution program.
Use an appropriate function to provide relevant debugging information for the following types of errors...

- When a FACS error occurs in your program, call one of the error functions described in this chapter, so that the record type, ordinal number, and the value of R0 are dumped.

When a **"find" error** occurs when your program is attempting to retrieve a record from DASD, call one of the error functions so that the SUD byte and file address reference word are dumped.

Error Processing Skill Check Solution

```
/*(C) Copyright Galileo International     */
/*Partnership, 1998. All rights rserved. */
/* This trb1 program receives input from the user indicating     */
/* whether the program is to display booked items in a specified */
/* PNID record, or cancelled items in the specified PNID record, */
/* whether a new name is to be added to the record, or if a new  */
/* chain is to be added to a PNID prime record.                   */
/* The PNID record is specified via the PNID ordinal number that */
/* must be provided as input, following a secified B or C input  */
/* to indicate whether booked or cancelled items will be listed, */
/* or an N to indicate that a name is to be added to the PNID,   */
/* or an A to indicate that a chain is to be added to the prime. */
/* Input must match one of the following formats:                */
```

```
/*                                                                    */
/*               t*trb1/b xxxx                                        */
/*                                                                    */
/*                   or                                               */
/*                                                                    */
/*               t*trb1/c xxxx                                        */
/*                                                                    */
/*                   or                                               */
/*                                                                    */
/*               t*trb1/n xxxx ppppppppppp                            */
/*                                                                    */
/*                   or                                               */
/*                                                                    */
/*               t*trb1/a xxxx                                        */
/*                                                                    */
/*                                                                    */
/* In the above input formats, b and c indicate whether booked        */
/* or cancelled items are being searched for.  xxxx represents        */
/* the PNID record ordinal number. n indicates a new name is to       */
/* is to be added to the prime PNID indicated by the ordinal          */
/* number.  ppppppppppp represents the new name.                      */
/* The letter 'a' means add a chain to the prime if a chain           */
/* doesn't already exist.                                              */
/*                                                                    */
/* This program uses data levels D0, D4 and D5                        */
/*                                                                    */
/*                                                                    */

#include "gifirst.h"
#include <tpfeq.h>
#include <tpfio.h>
#include <tpfapi.h>
#include <stdio.h>
#include "gitpf.h"
#ifndef pd1_structure
#define pd1_structure 1

#pragma pack(1);                    /* remove pad bytes */
struct pd1hd              /*header and common area of pd1pd */
{
char pd1bid[2];          /*record id = pd */
char pd1chk;              /*record code check */
char pd1ctl;              /*control byte-bit4 on = chain */
char pd1pgm[4];          /*last program to file this rec */
unsigned long pd1fch;    /*forward chain */
short pd1inx;            /*location index */
char pd1id1;              /*indicator byte */
char pd1id2;              /*another indicator byte */
long pd1flt;              /*flight number, 3=regular,4=extra */
short pd1dat;            /*depart date of last seg - binary */
short pd1sp6;            /*aci indicator byte */
unsigned long pd1wlp;    /*active waitlist file addr */
unsigned long pd1wln;    /*cancelled waitlist file addr */
unsigned long pd1gri;    /*grid file address */
char pd1sp2[16];         /*spares */
};

struct pd1itm              /*structure for pnix item*/
{
unsigned int pd1id3_bit0: 1;        /*cancelled from pni */
unsigned int pd1id3_bit1: 1;        /*group block pnr */
unsigned int pd1id3_bit2: 1;        /*sold from group block pnr */
unsigned int pd1id3_bit3: 1;        /*pta pnr */
unsigned int pd1id3_bit4: 1;        /*name is truncated */
unsigned int pd1id3_bit5: 1;        /*corporate name */
unsigned int pd1id3_bit6: 1;        /*primary - 1st or only name */
unsigned int pd1id3_bit7: 1;        /*facts or supplementary info */
```

```
unsigned int pd1id4_all: 8;         /*control byte */
unsigned int pd1sp3_all: 8;         /*another indicator byte */
unsigned int pd1id5_all: 8;          /*indicators-formerly pd1scc */
unsigned char pd1cos;                /*class of service bcd */
unsigned long pd1add;                /*pnr address */
unsigned char pd1prc;                /*record check code */
char pd1name[12];                    /*passenger name */
char pd1bpt[3];                      /*board point */
char pd1opt[3];                      /*off point */
unsigned char pd1nua;                /*binary# of pass with surnam */
unsigned char pd1npt;                /*binary number in party */
unsigned char pd1bkt;                /*bucket number */
char pd1sp4;                         /*spare */
short pd1bdt;                        /*booking date */
short pd1tdt;                        /*ticket & cancel date */
short pd1kri;                        /*kri item # for sch chg */
char pd1mcc[2];                      /*marketing carrier code */
};

#define RECTYPE "#PNDRI "           /* define PNID record type */
/* define number of items per PNID */
#define NUM_ITM \
( (_LBSZE - sizeof (struct pd1hd) ) / sizeof (struct pd1itm) )

struct pd1pd         /* template of PNID record */
{
struct pd1hd d1_header;                /*header/common area*/
struct pd1itm d1_items[NUM_ITM];       /* items */
char pd1sp5[3];                         /* spares */
};
#endif
/* define LIMIT as number of possible items in PNID record */
#define LIMIT 25


#define BOOKED_HEADER "List of Booked Passengers:"
#define CANCELLED_HEADER "List of Cancelled Passengers:"

void add_chain(struct pd1pd * pnid_ptr);
void name_add(char * name, struct pd1pd * pnid_ptr);
void display(char request, struct pd1pd * pnid_ptr);
void errors(long error_num,long ordnum,char rtype[]);
void FACS(struct TPF_regs * regs);
void trb1(void)
{
    struct TPF_regs regs;        /* declare registers for FACS */
    struct mi0mi * mi_ptr;       /* ptr to input msg */
    struct pd1pd * pnid_ptr;     /* ptr to PNID record */
    struct pd1itm * item_ptr;    /* ptr to single PNID item */

    char new_name[13];             /* for adding a name */
    char request_type;            /* cancelled (C) or booked (B) */
    char junk;                    /* holds discarded B or C from input */
    long ord;                    /* PNID ord number */
    short num_of_args;           /* scanf return value */
/*****************************************************************/
/* Interrogate input message                                     */
/*****************************************************************/
mi_ptr = (struct mi0mi *) ecbptr()->ce1cr0; /* set ptr to msg blk */
request_type = mi_ptr->mi0acc[7];    /* is input B or C or N or A? */

/* The above use of the mi0mi structure is here just to show you
   that you can use the mi0mi structure to access the input message.
   Below, we use scanf() to get other parts of the input message.
   We have already accessed the B, C, N or A part of the input, so in
   the scanf() below, we assign the B, C, N or A to a junk variable
   and just discard it.  */
```

```
if ( request_type != 'B' && request_type != 'C' &&
        request_type != 'N' && request_type != 'A' )
{
   errors(1,0,"dummy parms");          /* input format error */
}

if ( request_type == 'N' )
{
   num_of_args = scanf("T*TRB1/ %c %ld %12s",&junk,&ord,new_name);

   if ( num_of_args < 3 ) /* if input format error */
   {
      errors(1,0,"dummy parms");       /* input format error */
   }
}
else     /* B C or A type entry */
   num_of_args = scanf("T*TRB1/ %c %ld", &junk, &ord);
if ( num_of_args < 2 )
{
   errors(1,0,"dummy parms");          /* if input format error */
}
/*********************************************************************/
/* Setup registers for FACS                                          */
/*********************************************************************/
regs.r6 = (long) RECTYPE;
regs.r7 = (long) &(ecbptr()->ce1fa4);
regs.r0 = ord;

/*********************************************************************/
/* Call FACS                                                         */
/*********************************************************************/
FACS(&regs);
if ( regs.r0 == 0)               /* FACS error? */
{
    errors(2,ord,RECTYPE);        /* Yes, FACS error */
}
/*********************************************************************/
/* Retrieve the PNID prime record onto level 4.  HOLD or NOHOLD   */
/*depends on request type provided by user.                          */
/* If find fails, error function called.                             */
/*********************************************************************/
if ( request_type == 'N' || request_type == 'A' )
{
   pnid_ptr = find_record(D4,NULL,"PD",'\0',HOLD)
}

else                              /* B or C input */
   pnid_ptr = find_record(D4,NULL,"PD",'\0',NOHOLD)
if ( pnid_ptr == 0 )             /* find error? */
{
   errors(3,0,"dummy parms");    /* Yes, find error */
}


/*********************************************************************/
/* Call the display, name_add, or add_chain function,                */
/* depending upon input request.                                     */
/*********************************************************************/
switch ( request_type )
{
  case 'A':
           add_chain(pnid_ptr);
           break;
  case 'B':
           display(request_type, pnid_ptr);
           break;
```

```
  case 'C':
           display(request_type, pnid_ptr);
           break;
  case 'N':
           name_add(new_name, pnid_ptr);
           break;
}                  /* end of switch */

/*****************************************************************/
/* Release resources before program ends and have ECB exit system */
/*****************************************************************/
crusa(2,D0,D4);       /* release input block and PNID block */
exit(0);              /* release ECB and exit system */
return;
}                     /* end of trb1 function */
/*****************************************************************/
/* This is the add_chain function. It gets a pool record, sets up     */
/* the PNID header on the new chain, puts the new chain's file addrs  */
/* in the forward chain field of the PNID prime, files the new chain, */
/* files and unholds the updated prime.  Level D5 is used for chain.  */
*//*****************************************************************/
void add_chain(_Packed struct pd1pd * pnid_ptr)
{
  struct pd1pd * chain_ptr;
  unsigned long faddr;

  if ( pnid_ptr->d1_header.pd1fch == 0 )    /* if no exiting chain */
  {
    faddr = getfc(D5,GETFC_TYPE1,"PD",GETFC_BLOCK,GETFC_SERRC);
    chain_ptr = ecbptr()->ce1cr5;
    chain_ptr->d1_header = pnid_ptr->d1_header;
    chain_ptr->d1_header.pd1inx = sizeof(struct pd1hd);
    pnid_ptr->d1_header.pd1fch = faddr;
    memcpy(&ecbptr()->ebcid5,"PD",2);     /* put id in FARW */
    filec(D5);            /* file the new chain */
    filuc(D4);          /* file and unhold prime */
    puts("Chain added to prime");
  }
  else     /* prime already has chain */
  {
    puts("Chain already exists");
    unfrc(D4);     /* unhold the prime */
    relcc(D4);     /* release the prime block */
  }
return;
}     /* end of add_chain function */
/*****************************************************************/
/* This is the display function. It displays either booked or         */
/* cancelled passenger names from the PNID record, according to       */
/* the request type specified in the user input message.              */
/*****************************************************************/
void display(char request,struct pd1pd * pnid_ptr)
{
int index = 0;               /* looping index */
int cancelled = 0;          /* cancelled flag */
int booked = 0;             /* booked flag */
int count;                   /* string count for sprintf */
char buff[300];             /* buffer for sprintf */
char name_temp[13];         /* temp area for sprintf */
if ( request == 'B' )       /* look for booked items if B input */
{
     count = sprintf(buff,"%s n",BOOKED_HEADER);
     while ( index < LIMIT )
     {
          if ( pnid_ptr->d1_items[index].pd1id3_bit0 == 0 )
          {
            memcpy(name_temp,pnid_ptr->d1_items[index].pd1name,12);
```

```
            name_temp[12] = '\0';        /* add terminating null */

            count+=sprintf(buff+count,"%s\n",name_temp);

            booked = 1;          /* set flag */
          }           /* end of if */
        index++;                    /* bump to next item in record */
      }  /* end of while loop */
      printf("%s",buff);
      if ( booked == 0 ) /* Additional output if no booked items */
      {
          puts("NO BOOKED ITEMS FOUND");
      }   /* end of if */

} /* end of if */

if ( request == 'C' )
{
   count = sprintf(buff,"%s n",CANCELLED_HEADER);
   while ( index < LIMIT )
   {
       if ( (pnid_ptr->d1_items[index].pd1id3_bit0 ))
       {
          memcpy(name_temp,pnid_ptr->d1_items[index].pd1name,12);

          name_temp[12] = '\0';         /* add terminating null */

          count+=sprintf(buff+count,"%s\n",name_temp);

          cancelled = 1;            /* set flag */
       }
       index++;       /* bump to next item in record */

   }  /* end of while loop */
   printf("%s",buff); /* send output */

   if ( cancelled == 0 ) /* Additional output if no cancelled items */
   {
       puts("NO CANCELLED ITEMS FOUND");
   }   /* end of if */

}  /* end of if */
return;
} /* end of display function */


/*********************************************************************/
/* This is the name_add function. It places a passenger name         */
/* (provided as user input) in the next available item in the PNID.  */
/* The booked indicator is turned on, the location index is updated, */
/* and the PNID record is filed and unheld.                          */
/*********************************************************************/
void name_add( char * name, struct pd1pd * pnid_ptr )
{
  short length;                    /* For number of name letters */
  short k;                         /* index into PNID item array */
  k = (pnid_ptr->d1_header.pd1inx - sizeof(struct pd1hd))/
                 sizeof(struct pd1itm);
  length = strlen(name);

  while( length < 12 )             /* replace terminating null and */
  {                      /* and remaining array elements with spaces */
    name[length] = ' ';
    length++;
  }
```

```
  /* place name in PNID */
  memcpy(pnid_ptr->d1_items[k].pd1name, name, 12);

  pnid_ptr->d1_items[k].pd1id3_bit0 = 0;     /* set as booked */

  /* update location index */
  pnid_ptr->d1_header.pd1inx += sizeof(struct pd1itm);

  filuc(D4);            /* file and unhold prime PNID */

  puts("Name added to PNID");
  return;
}      /* end of name_add function */
/*****************************************************************/
/* This is the errors function. It displays messages back to the    */
/* user in response to input message format errors, FACS errors,    */
/* or record find errors detected in the program.                   */
/* For error_num == 2, the parms ordnum and *rtype are needed.       */
/* Those two parms are dummies for error_num == 1 or 3.              */
/*****************************************************************/
void errors( int error_num, long ordnum, char rtype[])
{
  struct snapc_list *cvsnap 3 , slist[3];
  switch (error_num )
  {
    case 1:        /* input format error */
          puts("trb1tg: una proc, input format error, abort");
          abort();
    case 2:       /* FACS failed */
          cvsnap[0] = &slist[0];           /*addressability*/
          cvsnap[1] = &slist[1];
          cvsnap[2] = &slist[2];

          cvsnap[0]->snapc_len = strlen (rtype);
          cvsnap[0]->snapc_name = "REC TYPE";
          cvsnap[0]->snapc_tag = rtype;
          cvsnap[0]->snapc_indir= SNAPC_NOINDIR;

          cvsnap[1]->snapc_len = sizeof (ordnum);
          cvsnap[1]->snapc_name = "ORDINAL#";
          cvsnap[1]->snapc_tag = &ordnum;
          cvsnap[1]->snapc_indir= SNAPC_NOINDIR;

          cvsnap[2]->snapc_len = 0;         /*end of list*/


          printf ("trb1tg: una proc, facs error, snapc");
          snapc (SNAPC_EXIT, 0X222222,
                   "FACS ERROR: REC TYPE, ORD # & REGISTERS",
                        cvsnap, 'U', SNAPC_REGS, SNAPC_ECB, NULL);
    case 3:         /* record retrieval failed */
          puts("trb1tg: una proc, find error, serrc_op");
          crusa(2,D0,D4);

          serrc_op (SERRC_EXIT, 0x333333,"FIND ERROR", NULL);
  }    /* end of switch */
} /* end of errors function */
```

CHAPTER 15

Accessing TPF Globals with TPF C

This chapter provides an introduction to accessing TPF globals when you are using TPF C.
Proceed through the chapter by reading all sections in order, beginning with the chapter objectives.

Objectives

Upon completion of this chapter, the participants will, with the aid of notes, be able to:

- Code the **glob()** function and obtain addressability to a global field

Tag Names for TPF Globals

TPF C uses tag names to identify specific global fields and records.
To access and modify individual global fields or records, the tag names are specified as parameters to the TPF C functions related to globals.
The **c$globz.h** header contains the tag names, and that header is included in the **tpfglbl.h** header.

Tag Name Format

Tag names are derived from corresponding assembler labels. However, the standard assembler global label @ character has been replaced with an underscore, due to C naming restrictions.
For example **@PTACT** global in assembler corresponds to the **_ptact** global tag name in TPF C.

Meaning of the Tag Name

Each tag name is assigned a unique 32-bit number, indicating various attributes, including the field or record's location in the global areas. The **c$globz.h** header contains the tag names and their corresponding 32-bit numbers.

Bits	Settings
0-3	Reserved by IBM (set to B'0000')
4-15	Displacement of item within its global area
16-23	If global field: field length If global record: record length in doublewords If synchronizable: SIGT slot number

24	0 = not keypointable 1 = keypointable
25	0 = not synchronizable 1 = synchronizable
26	0 = field 1 = record
27	Reserved by IBM
28	0 = field or record is not SSU common 1 = field or record is SSU common
29	Reserved for customer use
30-31	1 = data is in global area 1 3 = data is in global area 3

The TPF C Functions for Accessing Globals

The table below describes the primary TPF C functions related to accessing TPF globals.
Each C function in the table is a link to a section farther down on this page that describes the function in detail. Click on each function name to jump to that function's corresponding description, and then read that material so that you understand the function's purpose and syntax.

C Function	Purpose
glob()	Obtain the address of a global field or global record. The glob() function is a read-only function.
global()	Update, copy, keypoint, lock and unlock, or synchronize a global field. The global() function cannot be applied to global records.
glob_keypoint()	Keypoint a global field or record.
glob_lock()	Lock and access a synchronizable global field or record.
glob_unlock()	Unlock a global field or record.

glob_modify()	Modify a global field or record.
glob_update()	Update a global field or record.
glob_sync()	Synchronize a global field or record.

Below, let's examine some of the above TPF C functions...

All of the global functions are described in detail in IBM's TPF V4R1 C Language Support User's Guide in the Galileo ISO-C Bookshelf in BookManager.

glob() - Obtain the address of a global field or global record

Include	tpfglbl.h (includes c$globz.h)
Prototype	void * glob(unsigned int ***tagname***); The function returns the address of specified global field, or address of specified global record's global directory entry (which is the core address of the record).
Parameters	***tagname*** is the tag name of a particular global field or record.
Examples	`#include "gifirst.h"` `#include <tpfglbl.h>` `#include "gitpf.h"` `void pcx1(void)` `{` `  int *globfield_ptr;` `  struct psnec *rec_ptr;` `  ...` `  ...` `  /* Obtain addrs of global field */` `  globfield_ptr = (int *) glob(_ptact);` `  ...` `  ...` `  /* Obtain addrs of global record */` `rec_ptr = *(struct psnec **) glob(_psnec);` `  ...` `  ...` `}`
Usage Notes	1. This function provides read-only access to the specified global. 2. The tagname must be defined in c$globz.h. 3. 31-bit addressing is required to access globals in

	extended global areas.

global() - Copy, update, keypoint, lock, unlock, or synchronize a global field

Include	tpfglbl.h (includes c$globz.h)
Prototype	void *global(unsigned int ***tagname***, enum t_glbl ***action***, void * ***pointer***); Returned address varies depending upon parameters.
Parameters	***tagname*** is the tag name of a particular global field or record. ***action*** indicates the activity to be carried out. Use one of the following specifiers... **GLOBAL_UPDATE** updates the global field indicated by the tag name, with the value pointed to by the pointer parameter. Keypointing and synchronization are done if applicable. If field is to be synchronized, **LOBAL_LOCK** must be issued prior to **GLOBAL_UPDATE**. **GLOBAL_MODIFY** does the same as **GLOBAL_UPDATE**, except that keypointing and synchronization must be initiated through other calls. **GLOBAL_COPY** copies the value of the global field indicated by the tag name, into the location pointed to by the pointer parameter. **GLOBAL_KEYPOINT** keypoints the global field indicated by the tagname. The pointer parameter doesn't apply. **GLOBAL_LOCK** ensures exclusive use the global field indicated by the tag name. Core copy is refreshed by file copy. Then global field is copied to location pointed to by the pointer parameter. Valid only for synchronizable globals. Pointer parameter is required. Returns address of specified global field or directory entry. **GLOBAL_UNLOCK** releases exclusive use invoked by **GLOBAL_LOCK**. Requires previous issuance of the **GLOBAL_LOCK** action. The pointer parameter doesn't apply.

	GLOBAL_SYNC synchronizes the global field indicated by the tag name, across processors in a loosely coupled complex. Requires previous issuance of the **GLOBAL_LOCK** action. The pointer parameter doesn't apply. ***pointer*** is required for UPDATE, MODIFY, COPY and LOCK. Provide the address of source or destination, depending upon action.
Examples	`#include "gifirst.h"` `#include <tpfglbl.h>` `#include "gitpf.h"` `void pcx1(void)` `{` `  int ptact;` `  ...` `  ...` `  /* Have current global field value` `     copied into variable */` `  global(_ptact,GLOBAL_COPY,&ptact);` `  ptact = ptact+1; /* Update value */` `  /* Now update the global field */` `  global(_ptact,GLOBAL_UPDATE,&ptact);` `...` `...` `}`
Usage Notes	1. The tag name must be defined in **c$globz.h** header. Results are unpredictable, otherwise. 2. When issuing the lock, unlock, and sync actions, the program should not have any pending I/O operations outstanding, because these actions may perform the equivalent of a waitc(). 3. The issuance of a **GLOBAL_LOCK** should be followed quickly by a **GLOBAL_UNLOCK**, to minimize delay to other ECBs waiting for locks. 4. A program should not give up control between the reading and updating of a global, otherwise the data read may have already been updated by another program by the time the first program does its update. 5. The **global()** function cannot be applied to global records.

Global Skill Check

To test your knowledge of the TPF C global functions, you will be adding global-related functionality to the program solution you wrote for File I/O skill check 3, back in chapter 13.
Use the TPF C **global()** function to access the test system's current PARS date, via the **_u1day** global. Provide the PARS date as additional output header information, displayed on the booked and cancelled list entries handled by your program.
You will need to enter the assembler segment **UCDR** to convert the PARS date into a displayable form. Here's an example of how to enter UCDR for date conversion...

```
#include "gifirst.h"
#include <tpfeq.h>       /* needed for TPF_regs */
#include <tpfglbl.h>    /* needed for global functions */
#include "gitpf.h"
void pcx1(void)
{
   ...
   ...

   display(request,pnid_ptr);    /* call to display function */

   ...
   ...
}

void display(char request, struct pd1pd * pnid_ptr)
{
   unsigned short day;     /* variable for global value */
   struct TPF_regs regs;    /* for arguments to UCDR */

/* Use the global() function here, so the PARS date global
   value is placed in the day variable.  Call to global() not shown. */

/* Below, R6 is being set to the address of the location to which
/* UCDR is to return the converted date. */

   regs.r6 = (long) &(ecbptr()->ebw008);
   regs.r7 = (long) day;
   ecbptr()->ebw008 = 0xff; /* first byte must be non-zero */

   UCDR(&regs);          /* call to UCDR */

/* PARS date is now displayable... */

   printf("PARS Date: %s",&(ecbptr()->ebw008));
   ...
   ...
   ...
}
```

TPF Globals Skill Check Solution

```
/*(C) Copyright Galileo International    */
/*Partnership, 1998. All rights rserved. */
/* This trb1 program receives input from the user indicating     */
/* whether the program is to display booked items in a specified */
/* PNID record, or cancelled items in the specified PNID record, */
/* whether a new name is to be added to the record, or if a new  */
/* chain is to be added to a PNID prime record.                  */
/* The PNID record is specified via the PNID ordinal number that */
/* must be provided as input, following a secified B or C input  */
/* to indicate whether booked or cancelled items will be listed, */
/* or an N to indicate that a name is to be added to the PNID,   */
/* or an A to indicate that a chain is to be added to the prime. */
/* Input must match one of the following formats:                */
/*                                                               */
/*               t*trb1/b xxxx                                   */
/*                                                               */
/*                   or                                          */
/*                                                               */
/*               t*trb1/c xxxx                                   */
/*                                                               */
/*                   or                                          */
/*                                                               */
/*               t*trb1/n xxxx ppppppppppp                       */
/*                                                               */
/*                   or                                          */
/*                                                               */
/*               t*trb1/a xxxx                                   */
/*                                                               */
/*                                                               */
/* In the above input formats, b and c indicate whether booked   */
/* or cancelled items are being searched for.  xxxx represents   */
/* the PNID record ordinal number. n indicates a new name is to  */
/* is to be added to the prime PNID indicated by the ordinal     */
/* number.  ppppppppppp represents the new name.                 */
/* The letter 'a' means add a chain to the prime if a chain      */
/* doesn't already exist.                                        */
/*                                                               */
/* This program uses data levels D0, D4 and D5                   */
/*                                                               */
/*                                                               */

#include "gifirst.h"
#include <tpfeq.h>
#include <tpfio.h>
#include <tpfapi.h>
#include <stdio.h>
#include <tpfglbl.h>           /* for access to globals */
#include "gitpf.h"
#ifndef pd1_structure
#define pd1_structure 1

#pragma pack(1);                    /* remove pad bytes */
struct pd1hd              /*header and common area of pd1pd */
{
char pd1bid[2];          /*record id = pd */
char pd1chk;              /*record code check */
char pd1ctl;              /*control byte-bit4 on = chain */
char pd1pgm[4];          /*last program to file this rec */
unsigned long pd1fch;    /*forward chain */
short pd1inx;            /*location index */
char pd1id1;              /*indicator byte */
char pd1id2;              /*another indicator byte */
long pd1flt;              /*flight number, 3=regular,4=extra */
short pd1dat;            /*depart date of last seg - binary */
short pd1sp6;            /*aci indicator byte */
unsigned long pd1wlp;    /*active waitlist file addr */
unsigned long pd1wln;    /*cancelled waitlist file addr */
```

```
unsigned long pd1gri;    /*grid file address */
char pd1sp2[16];         /*spares */
};

struct pd1itm              /*structure for pnix item*/
{
unsigned int pd1id3_bit0: 1;        /*cancelled from pni */
unsigned int pd1id3_bit1: 1;        /*group block pnr */
unsigned int pd1id3_bit2: 1;        /*sold from group block pnr */
unsigned int pd1id3_bit3: 1;        /*pta pnr */
unsigned int pd1id3_bit4: 1;        /*name is truncated */
unsigned int pd1id3_bit5: 1;        /*corporate name */
unsigned int pd1id3_bit6: 1;        /*primary - 1st or only name */
unsigned int pd1id3_bit7: 1;        /*facts or supplementary info */
unsigned int pd1id4_all: 8;         /*control byte */
unsigned int pd1sp3_all: 8;         /*another indicator byte */
unsigned int pd1id5_all: 8;          /*indicators-formerly pd1scc */
unsigned char pd1cos;                /*class of service bcd */
unsigned long pd1add;                /*pnr address */
unsigned char pd1prc;                /*record check code */
char pd1name[12];                    /*passenger name */
char pd1bpt[3];                      /*board point */
char pd1opt[3];                      /*off point */
unsigned char pd1nua;                /*binary# of pass with surnam */
unsigned char pd1npt;                /*binary number in party */
unsigned char pd1bkt;                /*bucket number */
char pd1sp4;                         /*spare */
short pd1bdt;                        /*booking date */
short pd1tdt;                        /*ticket & cancel date */
short pd1kri;                        /*kri item # for sch chg */
char pd1mcc[2];                      /*marketing carrier code */
};

#define RECTYPE "#PNDRI "           /* define PNID record type */
/* define number of items per PNID */
#define NUM_ITM \
( (_LBSZE - sizeof (struct pd1hd) ) / sizeof (struct pd1itm) )

struct pd1pd         /* template of PNID record */
{
struct pd1hd d1_header;                 /*header/common area*/
struct pd1itm d1_items[NUM_ITM];       /* items */
char pd1sp5[3];                         /* spares */
};
#endif
/* define LIMIT as number of possible items in PNID record */
#define LIMIT 25


#define BOOKED_HEADER "List of Booked Passengers:"
#define CANCELLED_HEADER "List of Cancelled Passengers:"

void add_chain(struct pd1pd * pnid_ptr);
void name_add(char * name, struct pd1pd * pnid_ptr);
void display(char request, struct pd1pd * pnid_ptr);
void errors(long error_num,long ordnum,char rtype[]);
void FACS(struct TPF_regs * regs);
void trb1(void)
{
    struct TPF_regs regs;        /* declare registers for FACS */
    struct mi0mi * mi_ptr;       /* ptr to input msg */
    struct pd1pd * pnid_ptr;     /* ptr to PNID record */
    struct pd1itm * item_ptr;    /* ptr to single PNID item */

    char new_name[13];             /* for adding a name */
    char request_type;            /* cancelled (C) or booked (B) */
    char junk;                    /* holds discarded B or C from input */
```

```
    long ord;                    /* PNID ord number */
    short num_of_args;           /* scanf return value */
/*****************************************************************/
/* Interrogate input message                                     */
/*****************************************************************/
mi_ptr = (struct mi0mi *) ecbptr()->ce1cr0; /* set ptr to msg blk */
request_type = mi_ptr->mi0acc[7];    /* is input B or C or N or A? */

/* The above use of the mi0mi structure is here just to show you
   that you can use the mi0mi structure to access the input message.
   Below, we use scanf() to get other parts of the input message.
   We have already accessed the B, C, N or A part of the input, so in
   the scanf() below, we assign the B, C, N or A to a junk variable
   and just discard it.  */

if ( request_type != 'B' && request_type != 'C' &&
         request_type != 'N' && request_type != 'A' )
{
   errors(1,0,"dummy parms");          /* input format error */
}

if ( request_type == 'N' )
{
   num_of_args = scanf("T*TRB1/ %c %ld %12s",&junk,&ord,new_name);

   if ( num_of_args < 3 ) /* if input format error */
   {
      errors(1,0,"dummy parms");       /* input format error */
   }
}
else     /* B C or A type entry */
   num_of_args = scanf("T*TRB1/ %c %ld", &junk, &ord);
if ( num_of_args < 2 )
{
   errors(1,0,"dummy parms");          /* if input format error */
}
/*****************************************************************/
/* Setup registers for FACS                                      */
/*****************************************************************/
regs.r6 = (long) RECTYPE;
regs.r7 = (long) &(ecbptr()->ce1fa4);
regs.r0 = ord;

/*****************************************************************/
/* Call FACS                                                     */
/*****************************************************************/
FACS(&regs);
if ( regs.r0 == 0)               /* FACS error? */
{
    errors(2,ord,RECTYPE);      /* Yes, FACS error */
}
/*****************************************************************/
/* Retrieve the PNID prime record onto level 4.  HOLD or NOHOLD  */
/*depends on request type provided by user.                      */
/* If find fails, error function called.                         */
/*****************************************************************/
if ( request_type == 'N' || request_type == 'A' )
{
   pnid_ptr = find_record(D4,NULL,"PD",'\0',HOLD)
}

else                              /* B or C input */
   pnid_ptr = find_record(D4,NULL,"PD",'\0',NOHOLD)
if ( pnid_ptr == 0 )             /* find error? */
{
   errors(3,0,"dummy parms");    /* Yes, find error */
}
```

```
/*****************************************************************/
/* Call the display, name_add, or add_chain function,            */
/* depending upon input request.                                 */
/*****************************************************************/
switch ( request_type )
{
  case 'A':
           add_chain(pnid_ptr);
           break;
  case 'B':
           display(request_type, pnid_ptr);
           break;
  case 'C':
           display(request_type, pnid_ptr);
           break;
  case 'N':
           name_add(new_name, pnid_ptr);
           break;
}                 /* end of switch */

/*****************************************************************/
/* Release resources before program ends and have ECB exit system */
/*****************************************************************/
crusa(2,D0,D4);       /* release input block and PNID block */
exit(0);              /* release ECB and exit system */
return;
}                     /* end of trb1 function */
/*****************************************************************/
/* This is the add_chain function. It gets a pool record, sets up     */
/* the PNID header on the new chain, puts the new chain's file addrs  */
/* in the forward chain field of the PNID prime, files the new chain, */
/* files and unholds the updated prime.  Level D5 is used for chain.  */
*//*****************************************************************/
void add_chain(_Packed struct pd1pd * pnid_ptr)
{
  struct pd1pd * chain_ptr;
  unsigned long faddr;

  if ( pnid_ptr->d1_header.pd1fch == 0 )    /* if no exiting chain */
  {
    faddr = getfc(D5,GETFC_TYPE1,"PD",GETFC_BLOCK,GETFC_SERRC);
    chain_ptr = ecbptr()->ce1cr5;
    chain_ptr->d1_header = pnid_ptr->d1_header;
    chain_ptr->d1_header.pd1inx = sizeof(struct pd1hd);
    pnid_ptr->d1_header.pd1fch = faddr;
    memcpy(&ecbptr()->ebcid5,"PD",2);     /* put id in FARW */
    filec(D5);            /* file the new chain */
    filuc(D4);          /* file and unhold prime */
    puts("Chain added to prime");
  }
  else     /* prime already has chain */
  {
    puts("Chain already exists");
    unfrc(D4);     /* unhold the prime */
    relcc(D4);     /* release the prime block */
  }
return;
}     /* end of add_chain function */
/*****************************************************************/
/* This is the display function. It displays either booked or        */
/* cancelled passenger names from the PNID record, according to      */
/* the request type specified in the user input message.             */
/* Also the current PARS date is displayed on B or C input.          */
/*****************************************************************/
```

```
void display(char request,struct pd1pd * pnid_ptr)
{
int index = 0;              /* looping index */
int cancelled = 0;          /* cancelled flag */
int booked = 0;             /* booked flag */
int count;                  /* string count for sprintf */
char buff[300];             /* buffer for sprintf */
char name_temp[13];         /* temp area for sprintf */
unsigned short day;         /* for PARS date global */
struct TPF_regs regs;       /* for parms to UCDR */
global(_u1day,GLOBAL_COPY,&day);        /* global for PARS date */
regs.r6 = (long) &(ecbptr()->ebw008);
regs.r7 = (long)day;
ecbptr()->ebw008 = 0xff; /* first byte must be non-zero */
UCDR(&regs);        /* call UCDR to make PARS date displayable */
printf("PARS Date: %s",&(ecbptr()->ebw008));
if ( request == 'B' )       /* look for booked items if B input */
{
     count = sprintf(buff,"%s n",BOOKED_HEADER);
     while ( index < LIMIT )
     {
          if ( pnid_ptr->d1_items[index].pd1id3_bit0 == 0 )
          {
            memcpy(name_temp,pnid_ptr->d1_items[index].pd1name,12);

            name_temp[12] = '\0';        /* add terminating null */

            count+=sprintf(buff+count,"%s\n",name_temp);

            booked = 1;         /* set flag */
          }           /* end of if */
       index++;                    /* bump to next item in record */
     }  /* end of while loop */
     printf("%s",buff);
     if ( booked == 0 ) /* Additional output if no booked items */
     {
         puts("NO BOOKED ITEMS FOUND");
     }   /* end of if */

} /* end of if */

if ( request == 'C' )
{
   count = sprintf(buff,"%s n",CANCELLED_HEADER);
   while ( index < LIMIT )
   {
       if ( (pnid_ptr->d1_items[index].pd1id3_bit0 ))
       {
          memcpy(name_temp,pnid_ptr->d1_items[index].pd1name,12);

          name_temp[12] = '\0';         /* add terminating null */

          count+=sprintf(buff+count,"%s\n",name_temp);

          cancelled = 1;            /* set flag */
       }
       index++;       /* bump to next item in record */

   }  /* end of while loop */
   printf("%s",buff); /* send output */

   if ( cancelled == 0 ) /* Additional output if no cancelled items */
   {
       puts("NO CANCELLED ITEMS FOUND");
   }   /* end of if */

}  /* end of if */
```

```
return;
} /* end of display function */


/*********************************************************************/
/* This is the name_add function. It places a passenger name         */
/* (provided as user input) in the next available item in the PNID.  */
/* The booked indicator is turned on, the location index is updated, */
/* and the PNID record is filed and unheld.                          */
/*********************************************************************/
void name_add( char * name, struct pd1pd * pnid_ptr )
{
  short length;                    /* For number of name letters */
  short k;                         /* index into PNID item array */
  k = (pnid_ptr->d1_header.pd1inx - sizeof(struct pd1hd))/
                 sizeof(struct pd1itm);
  length = strlen(name);

  while( length < 12 )             /* replace terminating null and */
  {                       /* and remaining array elements with spaces */
    name[length] = ' ';
    length++;
  }
  /* place name in PNID */
  memcpy(pnid_ptr->d1_items[k].pd1name, name, 12);

  pnid_ptr->d1_items[k].pd1id3_bit0 = 0;     /* set as booked */

  /* update location index */
  pnid_ptr->d1_header.pd1inx += sizeof(struct pd1itm);

  filuc(D4);            /* file and unhold prime PNID */

  puts("Name added to PNID");
  return;
}      /* end of name_add function */
/*********************************************************************/
/* This is the errors function. It displays messages back to the     */
/* user in response to input message format errors, FACS errors,     */
/* or record find errors detected in the program.                    */
/* For error_num == 2, the parms ordnum and *rtype are needed.       */
/* Those two parms are dummies for error_num == 1 or 3.              */
/*********************************************************************/
void errors( int error_num, long ordnum, char rtype[])
{
  struct snapc_list *cvsnap 3 , slist[3];
  switch (error_num )
  {
    case 1:        /* input format error */
          puts("trb1tg: una proc, input format error, abort");
          abort();
    case 2:       /* FACS failed */
          cvsnap[0] = &slist[0];           /*addressability*/
          cvsnap[1] = &slist[1];
          cvsnap[2] = &slist[2];

          cvsnap[0]->snapc_len = strlen (rtype);
          cvsnap[0]->snapc_name = "REC TYPE";
          cvsnap[0]->snapc_tag = rtype;
          cvsnap[0]->snapc_indir= SNAPC_NOINDIR;

          cvsnap[1]->snapc_len = sizeof (ordnum);
          cvsnap[1]->snapc_name = "ORDINAL#";
          cvsnap[1]->snapc_tag = &ordnum;
          cvsnap[1]->snapc_indir= SNAPC_NOINDIR;

          cvsnap[2]->snapc_len = 0;         /*end of list*/
```

```
          printf ("trb1tg: una proc, facs error, snapc");
          snapc (SNAPC_EXIT, 0X222222,
                   "FACS ERROR: REC TYPE, ORD # & REGISTERS",
                        cvsnap, 'U', SNAPC_REGS, SNAPC_ECB, NULL);
    case 3:          /* record retrieval failed */
          puts("trb1tg: una proc, find error, serrc_op");
          crusa(2,D0,D4);

          serrc_op (SERRC_EXIT, 0x333333,"FIND ERROR", NULL);
  }    /* end of switch */
} /* end of errors function */
```

CHAPTER 16

Introduction to TPF C Dump Analysis.

This chapter provides an introduction to TPF C dump Analysis.
Proceed through the chapter by reading all sections in order, beginning with the chapter objectives.

Objectives

The purpose of this chapter is to introduce you to TPF C Dump Analysis.
This chapter is only an introduction. After you have had some TPF C experience and looked at a couple of TPF C dumps, you will want to take the "Advanced TPF C Dump Analysis" course, offered by Galileo International Technical Training.
After completing this chapter, the student will be able to:

- Identify the dump pointer to the writable static area.
- Identify the 3 Key registers and their usage in TPF C dumps.
- Identify the sections of a TPF C dump
- Find the automatic variables in the stack frames of a TPF C dump
- Find the parameter lists in the stack frames of a TPF C dump.
- Find the static variables in the writable static area of a TPF C dump.

Example Dump

Before you get into this chapter, you need to get copies of the example dump, DLM Listing, PCX2 Listing and PCX3 Listing. Contact a course manager to get your copies.

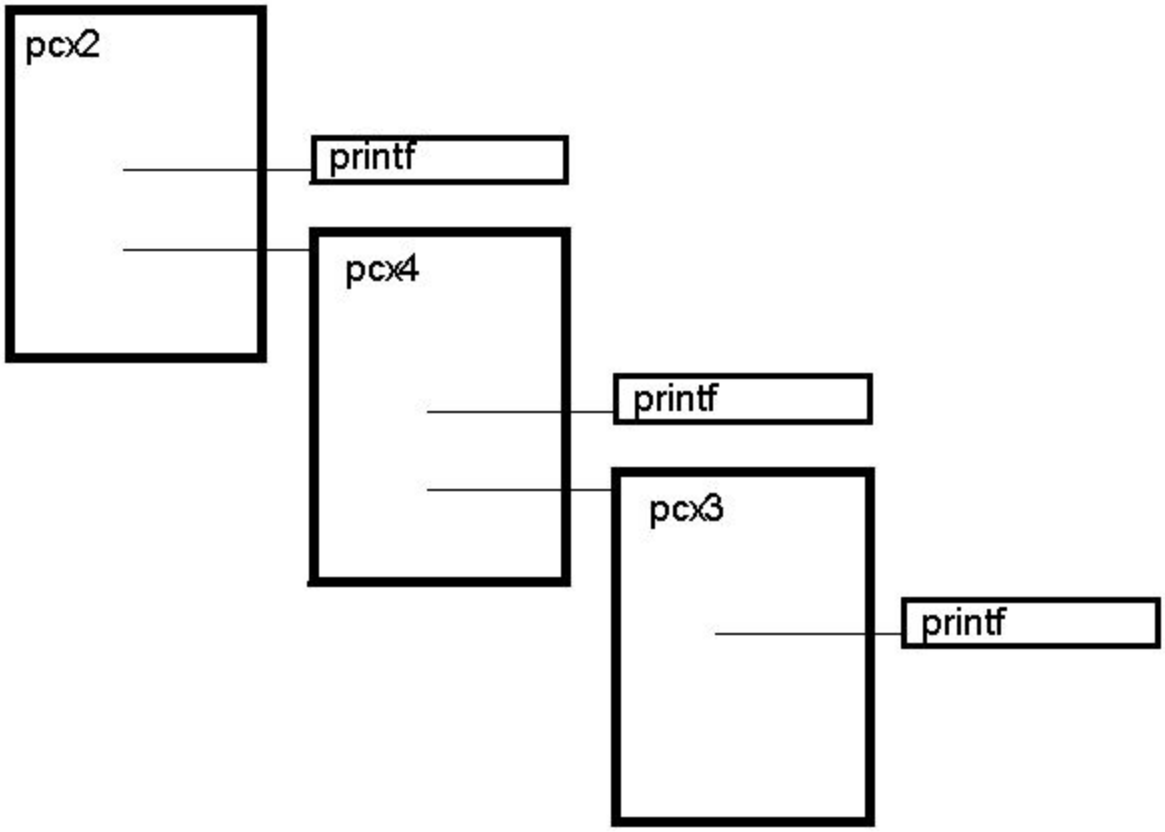
Explanation of the Example Dump

Programs **pcx2tv**, internal function **pcx4** within pcx2, and **pcx3tv** provide the vehicle for this chapter's dump example. These functions were built into **DLM pcx2t1**. These DLM was ran on the TASTE TAPOV01 test system. TASTE TAPOV01 is also where **dump #7111** was generated.
The logic flow was designed to create a **dump with stack frames and static variables.** The functions themselves perform **no rational purpose**, they are to provide you with an example of a dump with desired features. The functions declare automatic and static variables, call the puts and printf library functions, assign values to the variables, then call the next function with parameters.
Processing ends in pcx3. An **intentional protection exception** is caused by attempting to assign a value to a pointer initialized to NULL.
Hard copies of the **program and dump listings** are provided by the Course Manager. Irrelevant data in the listings may have been deleted to reduce bulk and wasting paper.
Analyzing the **pseudo assembler code** of the compiled program listing is useful in the dump analysis process. The pseudo code shows the instructions that actually execute, and the registers and displacements used.
Program listings used in this chapter were generated with the **no optimization** option to more easily understand the code.

The pages for the listings and dump have hand written pages on them. These page numbers are provided for ease of referencing the four documents as one...

Example Dump Program Flow

This diagram depicts the program flow of the example programs.
A stack frame is activated when each function is called, and is deactivated when the function returns. This includes library functions.
A released or deactivated stack frame may be reused by TPF for a subsequently activated function. The moral here is that once a function executes a return, the Stack Frame is no longer accessable.
Each of the 3 functions call printf, but the printf Stack Frames have come and gone before the dump takes place.
Functions PCX2 and PCX4 are located in Source Module PCX2. Function PCX3 is in Source Module PCX3.



Example Dump Program Variables and constants

Parameters are found in the Stack Frame of the calling function.
Declared auto variables are found in the function's Stack Frame.
Declared Static Variables and Control Strings used by printf can be found in the Writeable Static Area (WSP) for each function.

```
pcx2 (void)
```

```
          static char stat_char2;
                          /* Declared static */

          char auto_char2; /* Declared auto */

          printf("dump analysis exercise\n"
                 "stat_char2=%c, auto_char2=%c",
                     /*Control String - Static*/
                 stat_char2,  /* parameter */
                 auto_char2); /* parameter  */

          pcx4(stat_char2,    /* parameter  */
               auto_char2);   /* parameter  */


pcx4 (char p4_stat2,     /* Incoming parameter  */
      char p4_auto2)      /* Incoming parameter  */

          static short stat_int4;
                           /* Declared static */

          short auto_int4;  /* Declared auto */

          printf ("this is pcx4\n"
                 "stat_int4=%hd, p4_auto2=%c\n"
                 "stat_int4=%hd, auto_char4=%c",
                     /*Control String - Static*/
                 p4_stat2,    /* parameter  */
                 p4_auto2,    /* parameter  */
                 stat_int4,   /* parameter  */
                 auto_char4); /* parameter  */

          extern_pcx3 (stat_int4,
                       auto_int4);
                              /* parameters */


pcx3 (short p3_stat4,     /* Incoming parameter  */
      short p3_auto4)      /* Incoming parameter  */

          stat char stat3_char;
                            /* Declared static*/

          char auto3_char;   /* Declared auto */

          char * dump_ptr;   /* Declared auto */

          printf ("dis is pcx3\n"
                  "p3_stat4=%hd, p3_auto4=%hd\n"
                  "stat3_char=%c, auto3_char=%c",
                     /*Control String - Static*/
                  p3_stat4     /* parameter   */
                  p3_auto4     /* parameter   */
                  stat3_char   /* parameter  */
                  auto3_char); /* parameter  */
```

Example Program Input and Output

```
>t*pcx2
dump analysis exercise
stat_char2=A, auto_char2=B+?
this is pcx4
p4_stat2=D, p4_auto2=B
stat_int4=17, auto_int4=18+?
dis is pcx3
p3_stat4=20, p3_auto4=18,
stat3_char=Y, auto3_char=Z+?
>
```

Example Program Source Code

pcx2tv

```
01 /* (c)Copyright Galileo
International      */
02 /* Partnership, 1995. All Rights
Reserved. */
03
/*
 */
04 /* pcx2tv chapter 16, dump
analysis         */
05 /* function pcx2 calls function pcx4
which */
06 /* calls function pcx3tv to demonstrate
parms */
07 /* passed and stack and static in
core.    */
08 /* pcx3 dies with a protection
exception.  */
09
10 #include "giFIRST.h" /*galileo
standard  */
11 #include <stdio.h> /*for puts and printf
*/
12 #include <stdlib.h> /*for exit */
13 #include "giLAST.h" /*galileo standard
```

```
*/
14 #pragma map (extern_pcx3, "PCX3")
15
16 short PCX3 (short, short); /*prototype*/
17 static short pcx4 (char, char);
/*prototype*/
18
19 void pcx2 (void); /*prototype*/
20
21 void pcx2 (void)
22 {
23 static char stat_char2;
24 char auto_char2;
25
26 stat_char2 = 'A';
27 auto_char2 = 'B';
28 printf ("dump analysis exercise n"
29 "stat_char2=%c, auto_char2=%c",
30 stat_char2, auto_char2);
31
32 pcx4 (stat_char2, auto_char2);
33 exit (0);
34 }
35
36 short pcx4 (char p4_stat2, char p4_auto2)
37 {
38 static short stat_int4;
39 short auto_int4;
40
41 stat_int4 = 17;
42 auto_int4 = 18;
43 p4_stat2 = p4_auto2 + 2;
44
45 printf ("this is pcx4 n"
46 "p4_stat2=%c, p4_auto2=%c n"
47 "stat_int4=%hd, auto_int4=%hd",
48 p4_stat2, p4_auto2, stat_int4,
```

```
auto_int4);
49
50 extern_pcx3 (stat_int4, auto_int4);
51 return (0);
52 }
```

pcx3tv

```
01 /* (c)Copyright Galileo International */
02 /* Partnership, 1995. All Rights Reserved. */
03 /* */
04 /* pcx3tv chapter 16, dump analysis */
05 /* function pcx2tv calls function pcx4 which */
06 /* calls function pcx3tv to demonstrate parms */
07 /* passed and stack and static in core. */
08 /* pcx3 dies with a protection exception. */
09
10 #include <giFIRST.h>
11 #include <stdio.h>
12 #include <giLAST.h>
13
14 short pcx3 (short p3_stat4, short p3_auto4)
15 {
16 static char stat3_char;
17 char auto3_char;
18 char * dump_ptr = NULL;
19
20 stat3_char = 'Y';
21 auto3_char = 'Z';
22 p3_stat4 += 3;
23
24 printf ("dis is pcx3 n"
25 "p3_stat4=%hd, p3_auto4=%hd, n"
26 "stat3_char=%c, auto3_char=%c",
27 p3_stat4, p3_auto4, stat3_char, auto3_char);
28
29 *dump_ptr = 'F'; /* protection exception*/
30 return (0);
31 }
```

Common Sections of Assember and TPF C Dumps

All TPF dumps (assembler and TPF C) contain the sections described below. If you have worked with assembler dumps, you should already be familiar with these sections, and how they are used in dump analysis.
The listed page numbers refer to the example dump. Take a few minutes to look through the example dump to familiarize yourself with the various headings on the listed pages.

```
SYSTEM ERROR NUMBER

TRANSACTION PROCESSING FACILITY 4.1 PUT LEVEL 02

FAILING PROGRAM PCX2T1 = 0738 LOADSET-TESTCLS

STORAGE PROTECTION KEY
```

Page 3

```
IS STATUS DISPLAY

GENERAL REGISTERS

CURRENT PSW
```

Page 4

```
AREA OF PROGRAM ERROR
```

Page 8

```
AREA REFERENCED BY REGISTER ..
```

Page 30

```
COLLATED ECB MACRO TRACE ENTRIES
```

Page 47

```
COLLATED LDEVBK I/0 TRACE ENTRIES
```

Page 52

```
PREFIX PAGE
```

Page 54

```
BRANCH TRACE TABLE
```

Page 57

```
ENTRY REFERENCED VIA R9 -

DUMP OF ECB VIRTUAL MEMORY FOLLOWS
```

Page 63

```
PERM CORE PROG PCX2T1 LOADSET - TESTCLS
```

Unique Sections TPF C Dumps

TPF C dumps have the following unique sections.

These sections are different than what you have seen in assembler dumps. These are the sections that we will concentrate on for the rest of the chapter.
The listed page numbers refer to the example dump. Take a few minutes to look through the example dump to familiarize yourself with the various headings on the listed pages.

Page 64

```
ISOC CONTROL AREA
```

This section gives you the core address of the WSP (Writeable Static Area).

Look at the example dump for the Dump Label WSP:

```
. . .          . . .
. . .          04101010 WSP
. . .          . . .
```

Page 66

```
ISOC INITIAL STACK FRAME
```

The intial stack frame is the stack frame associated with the TPF startup code located in CSTRTD. When the TPF C environment was initialized for this ECB.

```
ISOC LWS
```

```
ISOC STACK FRAME FUNCTION . . .
```

These Stack Frames start on page 66 and continue to pages 67. You will have one Stack Frame for each active function. Usually there are only 2 or 3 active functions.

The Function Stack Frames exist only as long as the function. As soon as a function returns to a calling function, the called function's Stack Frame is released.

The Active Stack Frame is shown first (Last In First Out order). Then the following Stack Frame is for the function that called the current function. The next Stack Frame is for the function that called the previous Stack Frame's function, and so on.

In each Stack Frame, you will find the following Dump labels:

BKP - This is the back pointer. It points to the Stack Frame of the calling function.

NAB - This points to the called function's Stack Frame or the next available Stack Frame.

R14 and **R12** - These tags point you to the beginning and end of the register save area. These registers are saved by the function's Prolog.

Page 67

ISOC STATIC BLOCK

Each TPF C Source Module has it's own Static Area within the Static Block. Functions in a Source Module will share the Static Area for that Source Module.

IN USE HEAP STORAGE

There is no heap storage in use for the example dump. But, if there was it would show up under this Dump Label.

Heap Storage is TPF Memory accessed by malloc, calloc and realloc functions.

Writeable Static Area (WSP)

The WSP is a dump tag that contains the address of the Writeable Static Area (Static Block).
To find the WSP label, you will need to look in the ISOC Control Area (page 65) in the example dump.

ISOC Control Area

The Notable Dump Labels in the ISOC Control Area are:

BOS - Core Address of the beginning of the ISOC Stack Frames

EOS - Core Address of the end of the ISOC Stack Frames

WSP - Core Address of the Current writable static area

Registers of importance with TPF C Dump Analysis

First of all, forget about R8. With TPF Assembler, no segment can exceed 4K. With TPF C, a DLM can be as long as needed. Therefore, R8 may branch into a subsequent 4K block to execute a function and when the Dump occurs that subsequent block may not lead you to the correct conclusion about the cause of the Dump. Again FORGET ABOUT R8.
The three register to note and remember are:

R1 - points to the Function Parameter List
R13 - points to the Current Stack Frame
R15 - contains or points to the function Return Value.

R1

R1 is important, because it points to **the function parameter list**. So, when you call a function with parameters, R1 is used to point to the parameter list. In dump analysis, you will use the saved R1 in the function's stack frame to find the correct location of the parameter list. You cannot use the Dumped R1, because TPF C will use and reuse

registers as needed. Therefore, you must find a copy of R1 before potential corruption occurred. Luckily the PROLOG saves the registers for us at the top of the Stack Frame before turning control over to the function's code.
If you look in the example dump on page 66, find the ISOC Stack Frame for PCX3. Now, look over to find the Dump Label R14. This is the beginning of the register save area. Count from R14 over to find R1's register save area (R14, R15, R0, R1). It is 04001F90. This address is in the Current Stack Frame. If you find the address, you will find the list of parameters that function "pcx3" used to call the "printf" function.

R13

R13 is important, because it points to **the Current Stack Frame**. The dump register, R13, will point you to the Current Stack Frame. Find the Initial Stack Frame dump label (page 66) and compare R13 (page 1) to the first Stack Frame (page 66) after the Initial Stack Frame.

R15

R15 is the last register of importance. It is used by the EPILOG to point to **the Return Value**. If a function sends a Return Value, it is stored or pointed to by R15. Again, you cannot use the Dump Register because TPF C will probably have reused R15 before the dump occurred. This is what happened in the example dump.
However, you can verify the use of R15 if you go to hand written page 23. This is the Pseudo Assembly Listing for PCX2. If you look at displacement 0D6, you will see the "return (0);" command. Look at the Pseudo Assembler code for that command. It is a "LA r15,0".
The rest of the TPF registers can and will be used by TPF C for any number of reasons and/or functions. So, don't count on any of them to be a specific value.

Analyzing the CTL 3

There are 6 steps to resolving a control 3 dump. You will need your copy of the Example Dump and the TPF C Listings to follow allong with our example. Contact a Course Manager to get your copy.

1. The console log shows the ctl 3. (Hand written page 42)

```
CPSE0050E 15.49.16 IS-0001 RTL-A00039 UASU SE-007111
CTL-I000003
04018DA PCX2T1 00000738
PSW 071D2000 8142491C
R0-R7 000000BE 8128052C 040 . . .
```

2. The Dump shows the cause of error and identifies the failing program DLM and displacement. (Hand written page 42)

```
TAPE A00039 DUMP 7111 PAGE 1
SYSTEM ERROR NUMBER CTL-I000003
PROGRAM ERROR DETECTED IN OP
DATE - 17OCT TIME-15.49.16
CAUSE OF ERROR - PROTECTION EXCEPTION
*APPENDED CONSOLE MESSAGE - PSW 071D2000 8142491C
...
```

```
*FAILING PROGRAM PCX2T1 +00000738 LOADSET-TESTCLS
```

3. Now that we know the failing displacement, we can find the Dump Label "*PERM CORE PROG PCX2T1 LOADSET - TESTCLS" on handwritten page 64.
Start from the beginning Core Address for the DLM and then add the failing displacement.
This produces the address of the failing instruction.

> **Note: R8 is not necessarily the base address**, though it is in this example.
>
> 014241E0 DLM base Core Address
>
> + 00000738 Instruction Failing Displacement
>
> **= 01424918 Failing Instruction Address**
>
> (An alternate way of finding the failing instruction is thru the program old psw.)

Now find the address (hand written page 65 or 46).

```
*AREA OF PROG ERROR

. . .

014248F8 43605048 5060D0A4 1B664360 . 5860D094
01424918 92C66000 41F00000 58D0D004 . 01424990
```

Use your IBM Reference Summary to decode the Object Code.
This decodes to: **MVI 0(R6),X'C6'**

4. The Cross Reference Table output of the DLM build shows the starting displacement of the functions pcx2 and pcx3 in the DLM pcx2t1. Function pcx3 starts before the dump occurred and there is no function after pcx3. So the dump must have occurred in pcx3.
You can also use the Module Map to determine the function that failed. Our example is too old to include a Module Map.

```
          CROSS REFERENCE TABLE
CONTROL SECTION           ENTRY
NAME    ORIGIN  LENGTH    NAME       LOCATION

...
$PRIVATE   480      1F6
                          PCX2            480
$PRIVATE   678     104
                          PCX3            678
```

5. Now, we need to find the displacement into the pcx3 function.

So, we will subtract the failing displacement from the beginning of pcx3 function to get the displacement of the failing instruction, in pcx3.

```
  0738 failing displacement

- 0678 beginning of pcx3

= 00C0 failing displacement in pcx3
```

6. Now, we are ready to find the real line of TPF C code that failed. We know that the Dump occurred in function pcx3. So, we need to find the look at the Source Module Listing for the pcx3 function. The Source Module is PCX3TV. Go to the Pseudo Assembly code and find the failing displacement (00C0x). (Hand written page 37)

```
*
00029 | * *dump_ptr = 'F'; /*.*/
0BC 5860 D094 85 L r6,148(,r13)

0C0 92C6 6000 86 MVI 0(r6),198

00030 | * return (0);
```

The failing line of TPF C code is directly above the failing displacement pseudo assembly code. Look at line 29. We are trying to dereference the pointer "dump_ptr" and update the value to 'F'.
Now look at the Source section of PCX3TV Listing (hand written page 30) and find line 29 (*dump_ptr = 'F';). Look back through the code to see what the value of "dump_ptr" was when the dump occurred.
You will find that we initialized the pointer, but never pointed it at anything. Therefore, when we tried to dereference the pointer, we were trying to update low core (x'00000000'). That caused our Control 3.

Now we know what caused the dump. To fix the problem, we would make sure that "dump_ptr" was pointing to a valid location.

The stack frames are listed with the most current first, then backward through the calling function(s). Each stack frame is identified with it's associated function name.

This example will locate auto3_char declared in pcx3.

1. Locate the appropriate function's stack frame label in the dump. Hand written Dump pages 67 - 68, or the combined page 67/68. This page was created to allow you to look at the Stack Frames on one page.

```
*ISOC STACK FRAME FUNCTION-pcx3
. . .
*ISOC STACK FRAME FUNCTION-pcx4
. . .
*ISOC STACK FRAME FUNCTION-pcx2
. . .
```

(An alternate method is to find the R13, current stack frame pointer, on Dump page 1. The BKP pointer chains backward through the nested frames.)

```
R8-R15 014241E0 00500000 800E9596 00001000
04000010 04001EF8 81424914 00000040+
```

2. Determine the offset and length of a particular automatic variable from the Source Module Listing's STORAGE OFFSET LISTING. See PCX3TV listing and find Hand written page 40.

```
      * S T O R A G E    O F F S E T   L I S T I N *
IDENTIFIER  DEF    ATTRIBUTES
                    <SEQNBR>-<FILE NO>:<FILE LINE NO

auto3_char 17-0:17 Class=auto,   Offset=144,  Len=1

dump_ptr   18-0:18 Class=auto,   Offset=148,  Len=4

p3_auto4   14-0:14 Class=parm,   Offset=6,    Len=2

p3_stat4   14-0:14 Class=parm,   Offset=2,    Len=2

stat3_char 16-0:16 Class=static, Offset=72,   Len=1
```

3. Apply the offset and length of the automatic variable to beginning of the stack frame, to get the current value.

```
  04001EF8 beginning of pcx3 stack
+ 00000090 offset of auto3_char (144 dec)
```

= 04001F88 auto#_char = E9 = 'Z' (length Attribute is 1)

See Hand written Dump page 67

```
*ISOC STACK FRAME FUNCTION-pcx3
. . .
04001F58 00000000 00000000 00000000 00000000 00000
. . .
04001F78 00000000 00000000 04001EE0 00000000 E9000
BCD                                           Z
. . .
04001F98 00000012 000000E8 000000E9 1597F46D
BCD                     Y        Z      4
```

Automatic Variables Exercise

Using your copy of the Chapter 16 - Skill Check, complete the information for the Automatic variables only.

Finding a Parameter Variable in a TPF C Dump

Refresher on Parameter Variables

Parameter variables are like automatic variables. They reside in the Stack Frame area of core memory. However, they reside in the calling function's Stack Frame.
To find the parameter variables, you will use the value in the stored R1 in the calling function's Stack Frame. R1 will then point you to the list (in the calling function's Stack Frame).
Parameter variables can be either pointers or values. Depending on how information was passed (by value/copy or by reference/address).
They exist until the calling function calls another function, after the current function has returned.

Finding Parameter Variables in the Stack Frames of a TPF C Dump

This example locates **p3_stat4** in the current function, pcx3. It was passed by the calling function pcx4. pcx3 is the called function.

1. As a function is called, **R1** of the calling function points to the parameter list being passed. The parameter list is in the stack frame of the calling function.
The called function's prolog saves R14 - R7 in the calling function's stack.
Find the parameter list pointer, **R1 in the calling function's stack (pcx4)**.
Use the R14 register save area tag as a reference point.
See Hand written Dump page 67-68 or the combined page 67/68.

```
*ISOC STACK FRAME FUNCTION-pcx4

04001E48 .. 81424830 R14  01424858 ... 04001EE0

04001E68 ...
```

2. Find the offset (displacement) of a parameter in the storage offset listing, the PCX3TV Source Module Listing. See Hand written Page 40.

```
* S T O R A G E   O F F S E T   L I S T I N G *

. . .
p3_stat4 14-0:14 Class=parm, Offset=2, Len=2
```

3. Add the offset (displacement) of a parameter to the R1 pointer to find the parameter.

```
04001EE0 parameter list address from R1
+       2 offset from storage offset listing
04001EE2 address of p3_stat4
```

We can look at Hand written Dump page 68. Look for 04001EC8 (under Stack Frame for pcx4 - starts on page 67) and then count over to 04001EE0. Now add the 2 for displacement. We find a 0014 (attribute length is 2).

```
*ISOC STACK FRAME FUNCTION-pcx4
. . .
04001EA8 000 . . . 000 00000000 00000000 00000000
04001EC8 000 . . . 000 00000000 00000014 00000012
04001EE8 000 . . .                ^
                                  04001EE0
```

Parameter Variables Exercise

Using your copy of the Chapter 16 - Skill Check, complete the information for the parameter variables only.

Finding a Static Variable in a TPF C Dump

Refresher on Static Variables

Keyword is static. The keyword is required to define a static variable.
A static variable can be internal or external. Internal static variables are known only within the function where they are defined. So the scope is inside the curly braces ({ }). External static variables are known to all functions within the Source Module, below the variable's definition. So the scope is defined as within the Source Module.
static variables exist only for the Life of the ECB.
They are automatically initialized to x'00', if not explicitely initialized.
static variables live in the Heap Storage area core memory, more precisely in the Writable Static Area (WSP).

Finding Static Variables in the Writeable Static Area of a TPF C Dump

This example locates **stat_int4** in the shared pcx2-pcx4 static area.
The Function's Static Area base is the WSP address plus the offset for the Source Module. Once the beginning of the static area is determined, we will be ready to add the variable's displacement.

1. First, we must find the beginning of the DLM's Writable Static Area. So, we will look for the WSP in the ISOC Control Area on Hand written Dump page 65. Find the WSP Dump Label.

```
*ISOC CONTROL AREA
...
040001F0 07 ... 04101010 WSP      4E000000 ...
```

2. Now, we need to find the beginning of pcx4's Static Area.
Each Source Module in the DLM has it's own static area within the Writable Static Area.
To find the beginning of each Source Module's static area, we will use the DLM Listing File Map and Writable Static Map. The File Map tells us the Source Module number that we will cross reference in teh Static Map. The Static Map with the cross reference will identify each module's offset (displacement) into the Writable Static Area.
Since pcx2 and pcx4 are in the same Source Module, they share a static area.
To find the Source Module's offset, lets look at Hand written DLM Listing page 9. We will look for the File Map File ID for pcx2. It is a 00002.

```
==========================================

| File Map

==========================================

*ORIGIN  FILE ID   FILE NAME

PI        00001     DD:OBJLIB(CSTRTD40)
PI        00002     DD:OBJLIB(PCX2TV)
PI        00003    DD:OBJLIB(PCX3TV)
...
```

Now we can use the File Map File ID as a cross reference in the Writable Static Map. So, we will look for File ID 0002 and then we will find find the Source Module's offset, Again, look at Hand written DLM Listing page 9. The Offset for File ID 00002 is 50x. The Offset for File ID 00003 (PCX3TV) is 0.

```
==========================================

|                    Writable Static Map

+=========================================

OFFSET  LENGTH   FILE ID   INPUT NAME

     0      4C    00003     @STATIC
    50      80    00002     @STATIC
```

3. Next, we need to add our values together. So, the Source Module's Static Area Base = WSP + Offset for the Source Module.

```
  04101010 WSP
+       50 offset pcx2/pcx4 static area
= 04101060 base of pcx2/pcx4 static area
```

Up to now, everything has been in HEX. The Dump WSP is HEX and the Offset in the Writable Static Map is HEX.

4. Now, we will find the displacement from our Static Area Base for the variable. To do this, we need to go to the Source Module Listing's Storage Offset Listing. See Hand written page 26 to see PCX2TV's Storage Offset Listing. Find the line for stat_int4.

```
*S T O R A G E   O F F S E T   L I S T I N G*

...
stat_int4  38-0:38   Class=stat,  Offset=120,  Len=2
```

5. The last step is to add the Variable's Offset to the Static Area Base. We will then go find the variable in the Example Dump.

```
  04101060 static area base from above
+       78 offset (120 decimal)
= 041010D8 stat_int4 is x0011 = 17 (decimal)
```

:Now that we have the memory location of the variable, let's take a look. Look at Hand written Dump Page 68. You will find the ISOC Static Block. Under this Dump Label, you will find the beginning address of 04101000.
WAIT A MINUTE!!!! This address is not the same as the WSP.
Moral: don't rely on the address to begin exactly at the correct address.
If you look over x10 bytes (16 decimal), you will find the actual beginning of the Writable Static Area.
Now browse down to 041010C0 and count over to find 041010D8.

```
*ISOC STATIC BLOCK

04101000 D7C3E7F2 FFFFFF00 00000000 00000000 ...
. . .
04101060 84A49497 40819581 93A8A289 A24085A7 ...
04101080 F27E6C83 6B4081A4 A3966D83 888199F2 ...
041010A0 1597F46D A2A381A3 F27E6C83 6B4097F4 ...
041010C0 95A3F47E 6C88846B 966D8995 AF...    0011
```

Static Variables Exercise

Using your copy of the Chapter 16 - Dump Analysis Skill Check, complete the information for the static variables only.

CHAPTER 17

Miscellaneous TPF C functions

This chapter provides an introduction to a variety of TPF C functions.
The functions listed in this chapter are not as commonly used as functions listed in previous chapters.
You may want to scan the list of function on the various pages, but for most programmers, this material will only be useful for specific programming assignments. Therefore these functions would be more valuable at the time the assignment is coded. Note the functions available and read the details in the future.
Proceed through the chapter by reading all sections in order, beginning with the chapter objectives.

Objectives

The purpose of this chapter is to introduce you to a miscellaneous group of TPF C functions. After completing this chapter, the student will be able to code:

- Function calls for Creating a New ECB (creec, cremc, cretc, credc, cretc_level, crexc).
- Function calls for Handling Events (evntc, evnwc, evinc, evnqc, postc).
- Function calls for Delaying the ECB's processing (deferc, delayc).
- Function calls for Using Shared Resources (corhc, coruc, lockc, unlkc).

The TPF C Functions for Creating New ECBs

Some TPF Applications need to do Utility type processing. This processing requires the ability to create other ECBs. TPF C gives you the same ability as TPF Assembler to create these ECBs.
Basically each of the following functions is similar in nature to their Assembler cousins. The arguments are similar and they work in a similar manner. So, if you are familiar with the Assembler Create Macros, you shouldn't have any trouble with the TPF C Create Functions.

Reading this page

The table below describes the primary TPF C functions related to creating TPF ECBs. Each C function in the table is a link to a section farther down on this page that describes the function in detail. Click on each function name to jump to that function's corresponding description, and then read that material so that you understand the function's purpose and syntax.

C Function	Assembler macro equivalent	Purpose
creec()	CREEC	Create new ECB with an attached core block
credc()	CREDC	Create a deferred ECB

cremc()	CREMC	Create an immediate ECB
cretc()	CRETC	Create a time initiated ECB
cretc_level()	CRETC	Create a time initiated ECB with an attached core block
crexc()	CREXC	Create a low priority deferred ECB

Below, let's examine each of the listed TPF C functions...

creec() - Create new ECB with an attached core block

Include	tpfapi.h
Prototype	void creec(int length, const void *parm, void (*segname)(), enum t_lvl level, int priority); The function has no return.
Parameters	1. ***length*** is the number of bytes of data to be passed. The maximum number of bytes that can be passed is 104. 2. ***parm*** is the pointer to the data to be passed. 3. ***segname*** is the pointer to the external function to be called. 4. ***level*** is a valid data entry level. (D0 to DF) 5. ***priority*** determines if the new ECB will run immediately (CREEC_IMMEDIATE) or will run deferred (CREEC_DEFERRED).
Example	`#include <tpfeq.h>` `#include <tpfapi.h>` `...` `int value = 1345;` `...` **`creec(sizeof(value),&value,ABCD,D2,CREEC_IMMEDIATE);`** `...`
Usage Notes	1. The function issuing the CREEC may be forced to wait if there is insufficient storage to buffer the parameters. 2. Use of this function should be limited to prevent storage being depleted. 3. A system error occurs if more than 104 bytes are passed. 4. On return the block on the specified level is no longer available to the calling program. 5. The block is placed on LEVEL 0 of the new ECB.

credc() - Create a deferred ECB

Include	tpfapi.h
Prototype	void credc(int length, const void *parm, void (*segname)()); The function has no return.
Parameter	1. ***length*** is the number of bytes of data to be passed. The maximum number of bytes that can be passed is 104. 2. ***parm*** is the pointer to the data to be passed. 3. ***segname*** is the pointer to the external function to be called.
Example	`#include <tpfeq.h>` `#include <tpfapi.h>` `...` `int value = 1345;` `...` **`credc(sizeof(value),&value,ABCD);`** `...`
Usage Notes	1. The function issuing the CREDC may be forced to wait if there is insufficient storage to buffer the parameters. 2. Use of this function should be limited to prevent storage being depleted. 3. A system error occurs if more than 104 bytes are passed.

cremc() - Create an immediate ECB

Include	tpfapi.h
Prototype	void cremc(int length, const void *parm, void (*segname)()); The function has no return.
Parameter	1. ***length*** is the number of bytes of data to be passed. The maximum number of bytes that can be passed is 104. 2. ***parm*** is the pointer to the data to be passed. 3. ***segname*** is the pointer to the external function to be called.
Example	`#include <tpfeq.h>` `#include <tpfapi.h>` `...` `int value = 1345;`

	`` ... cremc(sizeof(value),&value,ABCD); ... ```
Usage Notes	1. The function issuing the CREMC may be forced to wait if there is insufficient storage to buffer the parameters. 2. Use of this function should be limited to prevent storage being depleted. 3. A system error occurs if more than 104 bytes are passed.

cretc() - Create a time initiated ECB

Include	tpfapi.h
Prototype	void cretc(int type, void (*segname)(), int units, const void *action); The function has no return.
Parameters	1. ***type*** is either seconds (CRETC_SECONDS) or minutes (CRETC_MINUTES). 2. ***segname*** is the pointer to the external function to be called. 3. ***units*** is an integer value representing the number of seconds or minutes before the entry is created. 4. ***action*** is a pointer to four bytes of information to be passed to the created ECB.
Example	``` #include <tpfeq.h> #include <tpfapi.h> ... cretc(CRETC_MINUTES,WXYZ,5,"NEW1"); ... ```
Usage Notes	1. Use of this function should be limited to prevent storage being depleted. 2. The TPF C cretc function does not allow a STATE parameter. 3. The four bytes specified to be passed will be placed at EBW000 on the created ECB.

cretc_level() - Create a time initiated ECB with an attached core block

Include	tpfapi.h
Prototype	void cretc_level(int type, void (*segname)(), int units, void *action, enum t_lvl level); The function has no return.
Parameter	1. ***type*** is either seconds (CRETC_SECONDS) or minutes (CRETC_MINUTES). 2. ***segname*** is the pointer to the external function to be called. 3. ***units*** is an integer value representing the number of seconds or minutes before the entry is created. 4. ***action*** is a pointer to four bytes of information to be passed to the created ECB. 5. ***leve***l is a valid data entry level. (D0 to DF)
Example	`#include <tpfeq.h>` `#include <tpfapi.h>` `...` **`cretc_level(CRETC_MINUTES,WXYZ,5,"NEW1",D4);`** `...`
Usage Notes	1. Use of this function should be limited to prevent storage being depleted. 2. The TPF C cretc_level function does not allow a STATE parameter. 3. The four bytes specified to be passed will be placed at EBW000 on the created ECB. 4. The data level passed will be placed on level 0 of the created ECB.

crexc() - Create a low priority deferred ECB

Include	tpfapi.h
Prototype	void crexc(int length, const void *parm, void (*segname)()); The function has no return.
Parameter	1. ***length*** is the number of bytes of data to be passed. The maximum number of bytes that can be passed is 104. 2. ***parm*** is the pointer to the data to be passed. 3. ***segname*** is the pointer to the external function to be called.
Example	`#include <tpfeq.h>` `#include <tpfapi.h>`

	`` ... int value = 1345; ... crexc(sizeof(value),&value,ABCD); ... ```
Usage Notes	1. The function issuing the CREXC may be forced to wait if there is insufficient storage to buffer the parameters. 2. Use of this function should be limited to prevent storage being depleted. 3. A system error occurs if more than 104 bytes are passed. 4. This function is similar to the credc function except that it is a lower priority to TPF and therefore requires a greater number of available core blocks before TPF will create the ECB. This function should be used in programs that create multiple ECBs. 5. The block is placed on LEVEL 0 of the new ECB.

The TPF C Functions for Handling Events

Some TPF Applications need to do Utility type processing. This processing requires the ability to create other ECBs. Some utilities require the created ECBs to communicate back to the originating ECB. This action is called an Event. TPF C gives you the same ability as TPF Assembler to handle created ECBs and their Events.
Basically each of the following functions is similar in nature to their Assembler cousins. The arguments are similar and they work in a similar manner. So, if you are familiar with the Assembler Event Macros, you shouldn't have any trouble with the TPF C Event Functions.

Reading this page

The table below describes the primary TPF C functions related to handling Events. Each C function in the table is a link to a section farther down on this page that describes the function in detail. Click on each function name to jump to that function's corresponding description, and then read that material so that you understand the function's purpose and syntax.

C Function	Assembler macro equivalent	Purpose
evntc()	EVNTC	Define an enternal event
evnwc()	EVNWC	Wait for completion of an internal event
evinc()	EVINC	Increment the count for a count type, internal event

evnqc()	EVNQC	Query the status of an internal event
postc()	POSTC	Post completion of an internal event element

Below, let's examine each of the listed TPF C functions...

evntc() - Define an internal event

Include	tpfapi.h
Prototype	int evntc(struct ev0bk *evninf, enum t_evn_typ evtype, char evn_name, int timeout, enum t_state evnstate); Returns a 0 value if processing completed normally. A '1' is returned if the event name already exists.
Parameters	1. ***evninf*** is a pointer to the event parameter block. 2. ***evtype*** is the type of event being defined; EVENT_MSK for mask events, EVENT_CNT for count events, EVENT_CB_Dn for core events on data levels 0 - F (replace the n with a valid data level). 3. ***evn_name*** is 'Y' if name is supplied in the event block. It is 'N' if the name is to be generated by TPF. 4. ***timeout*** is the number of seconds (0 to 65535 - half word) the ECB will wait before the event is considered to be in error. If zero is specified, the event will not timeout. 5. ***evnstate*** is the TPF environment state to allow the event to take place. If the event can run only in NORM state, use EVNTC_NORM. Otherwise, if the event can run in any TPF state, use EVNTC_1052.
Example	`#include <tpfapi.h>` `...` `struct ev0bk event_block;` `/* Set up the mask for the event */` `event_block.evnpstinf.evnbkml = 0xFFC0;` `...` `/* Define event wit 100 second timeout value,` `that can run in any state */` **`evntc(&event_block,EVENT_MSK,'N',100,EVNTC_1052);`** `...`
Usage Notes	1. If subsequent processing requires a completed event, you should use the evnwc() function to insure that the event has completed. Otherwise, control may return to the program prior to the completion of the event.

evnwc() - Wait for completion of an internal event

Include	tpfapi.h
Prototype	int evnwc(struct ev0bk *evninf, enum t_evn_typ evtype); Returns a 0 value if processing completed normally. A '1' is returned if the event has an error posted.
Parameter	1. ***evninf*** is a pointer to the event parameter block. 2. ***evtype*** is the type of event being defined; EVENT_MSK for mask events, EVENT_CNT for count events, EVENT_CB_Dn for core events on data levels 0 - F (replace the n with a valid data level).
Example	`#include <tpfapi.h>` `...` `struct ev0bk event_block;` `...` `/* Wait on an event of mask type to complete */` **`evnwc(&event_block,EVENT_MSK);`** `...`
Usage Notes	1. The evnwc parameter block will contain the remaining mask or count value, the accumulated POST MASK 2 for the event, and, if an error has occurred, the error indicator. 2. If the event is a core block event, the CBRW on the data level specified will reference the block pased by the postc.

evinc() - Increment the count for a count type, internal event

Include	tpfapi.h
Prototype	int evinc(struct ev0bk *evninf); Returns a 0 value if processing completed normally. A '1' is returned if the event cannot be found.
Parameter	1. ***evninf*** is a pointer to the event parameter block.
Example	`#include <tpfapi.h>` `...` `struct ev0bk event_block;` `...` `/* Increment count for a count event */`

	evinc(&event_block); ...
Usage Notes	1. The evinc() function increments the count for a count type event, after the event has been defined with the evntc() function. 2. An evnwc() function call cannot have been issued for the event before the evinc() call for the event.

evnqc() - Query the status of an internal event

Include	tpfapi.h
Prototype	int evnqc(struct ev0bk *evninf, enum t_evn_typ evtype); Returns a EVNQC_COM if the event completed. Returns a EVNQC_INC if the event is not completed. Returns a EVNQC_NFC if the event is not found.
Parameter	1. ***evninf*** is a pointer to the event parameter block. 2. ***evtype*** is the type of event being defined; EVENT_MSK for mask events, EVENT_CNT for count events, EVENT_CB_Dn for core events on data levels 0 - F (replace the n with a valid data level).
Example	`#include <tpfapi.h>` `...` `struct ev0bk event_block;` `...` `/* Query the status on an event of mask type */` **`evnqc(&event_block,EVENT_MSK);`** `...`
Usage Notes	1. The envqc parameter block will contain the remaining mask or count value, the accumulated POST MASK 2 for the event, and, if an error has occurred, the error indicator. 2.

postc() - Post completion of an internal event element

Include	tpfapi.h
Prototype	int postc(struct ev0bk *evninf, enum t_evn_typ evtype, int errcode);

	Returns a 0 value if processing completed normally. A '1' is returned if the event cannot be found.
Parameter	1. ***evninf*** is a pointer to the event parameter block. 2. ***evtype*** is the type of event being defined; EVENT_MSK for mask events, EVENT_CNT for count events, EVENT_CB_Dn for core events on data levels 0 - F (replace the n with a valid data level). 3. ***errcode*** is the error code. '0' if no errors, otherwise 1 to 256 is available as the range of error codes.
Example	`#include <tpfapi.h>` `...` `struct ev0bk event_block, *ev_blk_ptr;` `/* Event name and mask were passed from creating` `ECB at EBW000 */` `ev_blk_ptr = &(ecbptr()->ebw000);` `...` `/* Post completion of an event of mask type.` `Record no errors */` **`postc(ev_blk_ptr,EVENT_MSK,0);`** `...`
Usage Notes	1. For core block events, the CBRW on the data level specified will be marked as not holding a block. 2. For mask events, the postc parameter block must contain a 16-bit (half word) mask to be converted to a 1's complement and ANDed (&) against the mask value in the named event. • For count events, the current value of the event is automatically decremented by 1 when the post request is issued.

The TPF C Functions for Delaying the ECB Processing

Some TPF Applications need to insure a longer processing time than TPF normally allows. TPF C gives you the same ability as TPF Assembler to delay a TPF Time Out (Control 10 Dump).
Basically each of the following functions is similar in nature to their Assembler cousins. The arguments are similar and they work in a similar manner. So, if you are familiar with the Assembler Delay Macros, you shouldn't have any trouble with the TPF C Delay Functions.

Reading this page

The table below describes the primary TPF C functions related to delaying the ECB processing to prevent a TPF Time Out.

Each C function in the table is a link to a section farther down on this page that describes the function in detail. Click on each function name to jump to that function's corresponding description, and then read that material so that you understand the function's purpose and syntax.

C Function	Assembler macro equivalent	Purpose
defrc()	DEFRC	Defer processing of the current ECB
dlayc()	DLAYC	Delay processing of the current ECB

Below, let's examine each of the listed TPF C functions...

defrc() - Defer processing of the current ECB

Include	tpfapi.h
Prototype	void defrc(void); The function has no return.
Parameters	1. None
Example	`#include <tpfapi.h>` `...` **`defrc();`** `...`
Usage Notes	1. The function places the ECB on the DEFERRED list of the CPU. 2. The ECB must wait until TPF activities are low enough to allow completion of processing. 3. If a TPF Record is being held by the ECB when the defrc function is issued, a Control D (system dump) with exit will occur. 4. The execution of this function resets the TPF 500 millisecond Time Out counter for this ECB.

dlayc() - Delay processing of the current ECB

Include	tpfapi.h
Prototype	void dlayc(void); The function has no return.

Parameter	1. None
Example	`#include <tpfapi.h>` `...` `dlayc();` `...`
Usage Notes	1. The function places the ECB on the INPUT List of the CPU. 2. Unlike the defrc function, this ECB will be treated with equal priority as other new ECBs. This ECB is just placed at the end of the INPUT Message List and will restart processing as soon as it reaches the top of the list. 3. TPF Records should not be held by a delaying ECB. Too many ECBs waiting for the held record could result in the depletion of TPF working storage blocks. 4. The execution of this function resets the TPF 500 millisecond Time Out counter for this ECB.

The TPF C Functions for Using Shared Resources

Some TPF Applications need to use common work areas of TPF 4.1. This is one method of sharing information between ECBs. TPF C gives you the same ability as TPF Assembler to use the common areas of TPF.
Basically each of the following functions is similar in nature to their Assembler cousins. The arguments are similar and they work in a similar manner. So, if you are familiar with the Assembler Shared Resources Macros, you shouldn't have any trouble with the TPF C Shared Resources Functions.

Reading this page

The table below describes the primary TPF C functions related to sharing common resources between ECBs.
Each C function in the table is a link to a section farther down on this page that describes the function in detail. Click on each function name to jump to that function's corresponding description, and then read that material so that you understand the function's purpose and syntax.

C Function	Assembler macro equivalent	Purpose
corhc()	CORHC	Exclusively hold an in-core resource
coruc()	CORUC	Unhold an in-core resource

lockc()	$LOCKC	Lock a resource so that access by another I-Stream is prohibited
unlkc()	$UNLKC	Unlock a resource previously locked by the lockc() function

Below, let's examine each of the listed TPF C functions...

corhc() - Exclusively hold an in-core resource

Include	tpfapi.h
Prototype	void corhc(void *resource); The function has no return.
Parameters	1. ***resource*** is a pointer to the shared resource.
Example	`#include <tpfapi.h>` `...` `char *resource = "ABCD";` `...` **`corhc((void *)resource);`** `...`
Usage Notes	1. Control is not returned to the issuing program until exclusive hold of the resource is obtained. 2. Other entries wanting the resource are queued while the resource is held. 3. The ECB must call the coruc() function to free the resource.

coruc() - Unhold an in-core resource

Include	tpfapi.h
Prototype	void coruc(void *resource); The function has no return.
Parameter	1. ***resource*** is a pointer to the shared resource.
Example	`#include <tpfapi.h>` `...` `char *resource = "ABCD";` `...` **`coruc((void *)resource);`**

	. . .
Usage Notes	1. Use of this function without a prior corhc() function call will result in a system error. The ECB will be exited.

lockc() - Lock a resource so that access by another I-Stream is prohibited

Include	tpfapi.h
Prototype	int lockc(void *lockword, enum t_lock_opt opt); The function returns a '0' if a lock on the shared resource was obtained. A '4' is returned if a lock on the shared resource exists in another I-Stream and LOCK_O_RETN is specified
Parameter	1. ***lockword*** is a pointer to two consecutive fullwords used for lock and trace. 2. ***opt*** is the option for how the lock is attempted. If LOCK_O_SPIN is coded, TPF will spin for 1 second. If the resource becomes available in that time frame, the resource will be locked for this ECB. If the spin completes and the resource is still not available, a catastrophic dump is taken. LOCK_O_RETN will return a condition code if the resource is already locked by another ECB or it will lock the resource for this ECB if the resource is available.
Example	`#include <tpfapi.h>` `void *lock_ptr;` `...` `char *lock_ptr = &ofresource;` `...` **`lockc(lock_ptr, LOCK_O_SPIN);`** `...`
Usage Notes	1. If the spin lock times out, a Control 571 (system dump) will be taken. 2. The lock specified by lockword must not already be held by this I-Stream, otherwise a Control 572 (system dump) will be taken. 3. You must ensure that this I-Stream is not holding another Lock, otherwise a lockout condition could occur. Likewise, control must not be given up once a lock is held.

unlkc() - Unlock a resource previously locked by the lockc() function

Include	tpfapi.h
Prototype	int unlkc(void *lockword); The function returns a '0' if a unlock on the shared resource was sucessful.
Parameter	1. ***lockword*** is a pointer to two consecutive fullwords used for lock and trace.
Example	`#include <tpfapi.h>` `void *lock_ptr;` `...` `char *lock_ptr = &ofresource;` `...` **`unlkc(lock_ptr);`** `...`
Usage Notes	1. The lock specified by lockword must be held by this I-Stream, otherwise a Control 573 (system dump) will be taken, and then the lock is unlocked.

HOW TO PACK

When C headers are built from existing assembler DSECTs, care must be taken to ensure that the C structs are packed. With previous compilers the only way to do this was to use the _Packed keyword. However, there is now a preprocessor directive, #pragma pack, that should be used instead.

There are two advantages to the use of the directive over the keyword:

1. **It is easier to use.** The directive is straightforward in its implementation, making It easy to ensure that structures, nested structures, and unions are all packed.
2. **It will ease the migration to C++.** The #pragma pack directive is supported in C++. The _Packed keyword is not.

The format and use of #pragma pack preprocessor directive are detailed in Section 2.8.14.27-1 of the IBM manual *OS/390 V2R9 C/C++ Language*

An important point to note is:

#pragma pack() and #pragma pack(reset) are not the same.

Take this example:

```
#pragma pack(packed)  //first pragma

...                   // 1-byte alignment
```

```
#pragma pack(packed)  // second pragma

struct mystruct1;     // 1-byte alignment

#pragma pack()        // third pragma

struct mystruct2;     // 4-byte alignment
```

The first pragma has no effect on either struct. The second pragma specifies that mystruc1 should have 1-byte boundary alignment. The third pragma specifies that mystruct2 should have full word alignment.

In a second example:

```
#pragma pack(packed)   // first pragma

...                    // 1-byte alignment

#pragma pack(packed)   // second pragma

struct mystruct1;      // 1-byte alignment

#pragma pack(reset)    // third pragma

struct mystruct2;      // 1-byte alignment !!!!
```

The second pragma specifies that mystruc1 should have 1-byte boundary alignment. The third pragma resets the alignment to whatever it was previously. In this case because of the first pragma the alignment was previously 1-byte. So mystruct2 will have 1-byte boundary alignment.

The second important point to remember is:

If your header contains a #pragma pack to change alignment you must always reset the alignment by the end of the header.

TPF CONCERN

The best place to go for information about the TPF specific concerns for C programmers is IBM. More specifically, we recommend the **TPF Application Programming** manual in BookManager. If you are using the LAN version of BookManager, it can be found on the Galileo ISO-C bookshelf as well as the TPF bookshelf.

The other excellent sources of detailed information for TPF C programmers are IBM's TPF C web sites. See IBM Documentation.

We do not want to get into the business of rewriting IBM's documentation for our intranet, but there are some topics that generate so many questions that we have decided to write up short discussions of these topics. You will find links to these topics on the navigation bar on the left side of this page.

Writable Static, Reentrancy, and LLMs

LLM is the acronym for Library Load Module. This term refers to the TPF libraries that use library scripts and ordinal numbers to resolve linkage. A second type of libraries, Dynamic Link Libraries (DLL), are now available on TPF. Writable static is not a concern with DLLs. For this reason and because DLLs are easier to use in many other ways, it is recommended that all new libraries be created as DLLs.
This discussion applies to LLMs only.
This page was compiled largely from various sections of the IBM **TPF Application Programming** and **OS/390 C/C++ Programming Guide** manuals.
Let's start this discussion by stating the rules:

Library functions that use writable static:

- must be compiled with the RENT (reentrant) compiler option,
- should be isolated in separate libraries that contain only functions that use writable static.

Library functions that do not use writable static:

- must be compiled with the NORENT compiler option (and must be naturally reentrant),
- should be isolated in separate libraries that contain only functions that do not use writable static.

DLM functions that use writable static:

- **must** be compiled with the RENT option
- can be in the same DLM as other functions that do not use writable static, although isolating them in separate DLMs will improve performance**.**

DLM functions that do not use writable static:

- are compiled here at Galileo with the RENT option
- can be in the same DLM as other functions that use writable static, although isolating them in separate DLMs will improve performance.

That done, now let's explain the reasons behind these rules and the consequences if they are not followed.

What is writable static?

In C language a variable can be declared as static. This means that the variable is not reinitialized every time the function containing it is called. An example of a static variable is a counter that increments every time a function is called.

```
{
   static short number_of_calls = 1;
   ++number_of_calls;
}
```

The variable number_of_calls, though confined in scope to the function where it is declared, has a static duration and is persistent for the life of the program. In TPF terms

that means the variable is initialized the first time that an ECB enters this LLM and persists for the life of the ECB.
The first time this function is called, the value of number_of_calls will be 1. When the function completes the value will have been incremented to 2. Since number_of_calls is declared as static, each time this function is called, number_of_calls will have the value it had the last time that this function completed.
Because number_of_calls behaves differently from garden-type automatic variables, the compiler must make special arrangements for this variable. The exact manner in which the compiler handles this variable depends on whether or not the component is compiled using the reentrant (RENT) compiler option.

Reentrancy

Reentrancy allows the same program machine code to be used concurrently and repeatedly. **If the RENT compiler option is used**, C achieves reentrancy by splitting your program into two parts. The first part, which consists of executable code and constant data, does not change during program execution. The second part may be altered in the course of the program. This part includes the stack frames and a piece of storage known as the writable static area, which contains all persistent data that can be altered. The variable number_of_calls would be stored in the writable static area.
The code part of the program consists of:

- executable instructions
- read-only constants
- global variables defined with "#pragma variable (NORENT)"
- string literals, if #pragma strings(readonly) is in effect

The modifiable writable static area consists of:

- program variables with the static storage class
- program variables receiving the extern storage class (global variables as well as variables defined with the "extern" keyword)
- strings*

Program variables that are not static or extern are stored in stack frames.

It is important to understand that the writable static area is only created if the RENT compiler option is used and that the compiler puts more than just static variables in writable static. In fact most strings* are placed in the writable static area.

Non-reentrancy

If the RENT compiler option is not used then your program will not have a writable static area. The variables and strings that would have been stored there are instead stored in the program itself. If any of those static or extern variables are updated, your program is now self-modifying or non-reentrant.
On the other hand, if those same variables are only read, and only the variable in your stack area are updated, then your code is naturally reentrant.

Stack frames vs. Writable Static

A stack frame is used during the duration of a function. The frame is retained when other functions are called and reused when the called function returns.

Unlike stack frames, writable static blocks remain in existence until the ECB exits. If a function in a DLM or library needs writable static, a static storage area is allocated for that DLM or library from ECB heap at the time of DLM or library startup. There is a separate writable static area for each DLM or library.

Special Problems for Libraries

If a library requires writable static, then the linkage for any function in the library uses secondary linkage in order to accommodate the writable static block. This additional overhead causes slower performance in these functions. This bears repeating, **all the functions in a library use secondary linkage and writable static if any one library function uses writable static.** This results in performance losses for any function in the library.
The problem goes beyond performance loss. In some cases the use of writable static in library functions can cause run-time errors. We have run into one particular example here at Galileo. For a detailed explanation of that problem and more information on this subject see The Writable Static Problem in C Libraries by Bill Ashby in IBM's TPF Newsletter, Summer 1998.

The Bottom Line

When the RENT option is used, the compiler places every string* in the writable static area. So even if you have diligently isolated all of your non-writable static functions in a separate library, if one of your library components is compiled with the RENT option and it contains a string constant, your entire non-writable static library now becomes a writable static library. Your library will now be subject to the performance and functional problems associated with writable static libraries.
What should you do to prevent this? Request that the Software management group mark all of the components in your non-writable static library as NORENT in MCF. Both PROASM and AMP look at this indicator and will compile these components accordingly.
TPF being what it is, this will only work if your components are naturally reentrant.

Naturally reentrant C code

In order to make C code naturally reentrant, you need to eliminate the need for writable static. The easiest way to make C code naturally reentrant, is to follow these guidelines:

- do not use the "static" keyword
- do not use "extern" variables
- do not use global variables (unless made constant through the use of the "#pragma variable (NORENT)" directive)
- do not write to string literals (bad coding in any situation)

* - Strings that are defined using either the "const" keyword or use the preprocessor directive "#pragma strings (string_name, readonly)" will not be placed in the writable static area.