TPF Software

zTPF Concepts & Programming

TPF Software Inc. Training Material

TABLE OF CONTENTS

1. Introduction to zTPF

1.1 zTPF Overview

- **zTPF** is a High-performance operating system specifically designed to provide high availability for demanding high volume, real-time transaction processing for mission critical e-business applications.

- **zTPF** Runs on and exploits IBM's z/Series servers, an infrastructure offering for transaction processing with high quality of service demands.

- **zTPF** has been developed on the architecture of **TPF**. TPF is widely used in Travel, Banking/Finance, and Public Sector industry segments.

- **zTPF** is well suited to business computing environments requiring:

  - **High reliability/availability** - For mission critical applications, downtime can affect losses of millions of dollars an hour, not only from revenue loss, but also from bad publicity and dissatisfied customers.

  - **Fast response time -** When response time is critical to help lower call center costs or enhance customer satisfaction, zTPF is designed to provide fast and consistent response across predictable and unpredictable peaks.

  - **Low Cost -** zTPF offers a low cost per transaction for high-volume real-time transactions. This capability alone represents a competitive advantage for delivery of business services in the marketplace.

  - **Efficiency** - zTPF supports IBM's 64-bit z/Architecture with the use of 64-bit real addresses and 64-bit virtual addresses enabling large-scale memory spaces permitting large applications and large memory tables.

  - **Open development environment** - zTPF also uses the GNU tool chain. zTPF can share applications, tooling, and development infrastructure with the most common open system available, Linux.

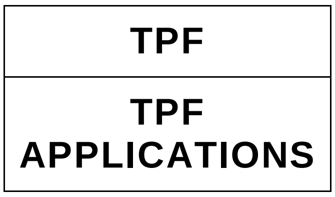
1.2 zTPF History

- **Early 1960's -** SABRE, PANAMAC
- **Mid 1960's** - ACP/PARS developed. First installations in US.
- **1970's -** ACP4, ACP6, ACP8, ACP9
- **1979** - TPF1
- **1983** - TPF2.1
- **1984** - TPF2.2
- **1985** - TPF2.3 (HPO, loosely coupled Support)
- **1988** - TPF2.4 (TCMP support)
- **1989** - TPF3.1
- **1993** - TPF4.1 announced (Delivered 1994).
- **1994 to 2005 -** PUT level 3 to PUT Level 20. Includes support for ISOC, support for TCP/IP, object oriented programming.
- **2005 September** - zTPF 64 Bit high-performance operating system released. zTPF runs on IBM zSeries servers.

1.3 Other Related Operating systems

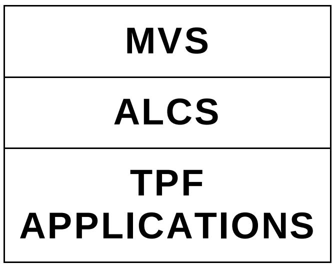
TPF

- TPF is an abbreviation to **Transaction Processing Facility.**
- TPF is a standalone operating system that runs on IBM Mainframes. TPF 4.1 is designed to run on S/390 based System Architecture.
- TPF is one of the major operating systems that runs under the S/390 Series of Mainframe Computers



ALCS

- Abbreviation for **Airline control System**.
- ALCS is a product designed to permit operation of TPF application programs under MVS with minimum modification.
- ALCS Version 2:
  - Operates in an MVS/ESA environment.
  - Includes 'C' language support.



1.4 zTPF Users across the world

Following are the few organizations that use ZTPF:

- **Airlines** – Continental Airlines, United Airlines, JAL, US Airways, KLM, Singapore Airlines etc.
- **Global Distribution Systems** – Sabre, Travelport
- **Hotels** – Marriot, Intercontinental Hotels Group
- **Credit Card processing/Banking** – American Express, VISA Inc.
- **Railways** - Amtrak

2. zTPF – Concepts and Conventions

2.1 Re-entrancy

- Reentrancy is to allow a single copy of the program to be used concurrently by two or more processes in a multi programming environment without disturbing the execution.

- To be reentrant, the application program must not modify any storage located within the bounds of the program itself. Instead, it must reference switches, indicators or counters in an application work area that is located in the ECB or owned exclusively by the entry.

  Eg: Defining a storage variable inside the program area which will violate the reentrancy concept as below.

Violates re-entrancy	Does not VIOLATE Re-entrancy
`$IS$   CSECT`	`$IS$   CSECT`
`      LHI    R1,1`	`       CALOC COUNT=R2,ESIZE=R3`
`       AHI    R1,10(R1)`	`      LHI    R1,1`
`       STH    R1,LABEL`	`       AHI    R1,10(R1)`
`           .`	`       STCM   R1,B'0011',0(R2)`
`           .`	`           .`
`           .`	`           .`
`LABEL DS    H`	`           .`
`           .`	`       LTORG`
`           .`	`       FINIS ABCD`
`           .`	`       END`
`       LTORG`	
`       FINIS ABCD`	
`       END`	

- This means that the program can suspend, abort, initiate, or resume processing of any number of entries at any point in their processing, without ever reinitializing.

- Reentrancy is required in a real time multi Programmed environment.

- Reentrancy in C and assembler programs is handled through the application stack, which is used during the duration of a function.

- The storage is retained when other functions are called and reused when the called function returns.

- The application stack contains the storage required by program variables unless the variables are declared to be static.
- Static data is ECB-specific, and changes to the data do not affect other ECBs using the program.

2.2 Entry & Transaction

- zTPF is an operating system in which most units of work are initiated by messages.
- The zTPF system is a conversational system in that a single response must be provided for each message.
- A process or task is usually triggered by the arrival of an Input message or ENTRY
- When an entry arrives at the system, zTPF allocates all the resources needed for its processing and tries to get back with a response.
- In the TPF Context Some Definitions
  - AgentSet – A zTPF Users Terminal from where entries are made
  - CRAS – Special Status Set which can issue system entries
  - LNIATA - Line Number, Interchange Address, Terminal Address.
  - Prime CRAS – System Console (PRC)
  - RO CRAS – System Receive Only Terminal
  - PARS – Airline Reservation System, Applications Suite
  - zTPF CP – zTPF Control Programs
- When an agent keys in an entry, it arrives as work to a zTPF system in the form of an “Input Message”.
- zTPF takes the block, constructs an **Entry Control Block** and dispatches the entry for processing.
- The relevant application/system programs are entered for processing.
- The application program issues API calls whenever needed for performing I/O and other operations.
- The databases are updated, if required and a response is generated and routed to the agent.
- Each entry should have a response being sent back to the terminal.
- This is mandatory because when an entry is keyed in, the keyboard remain locked (inactive) till a response clears the keyboard lock.

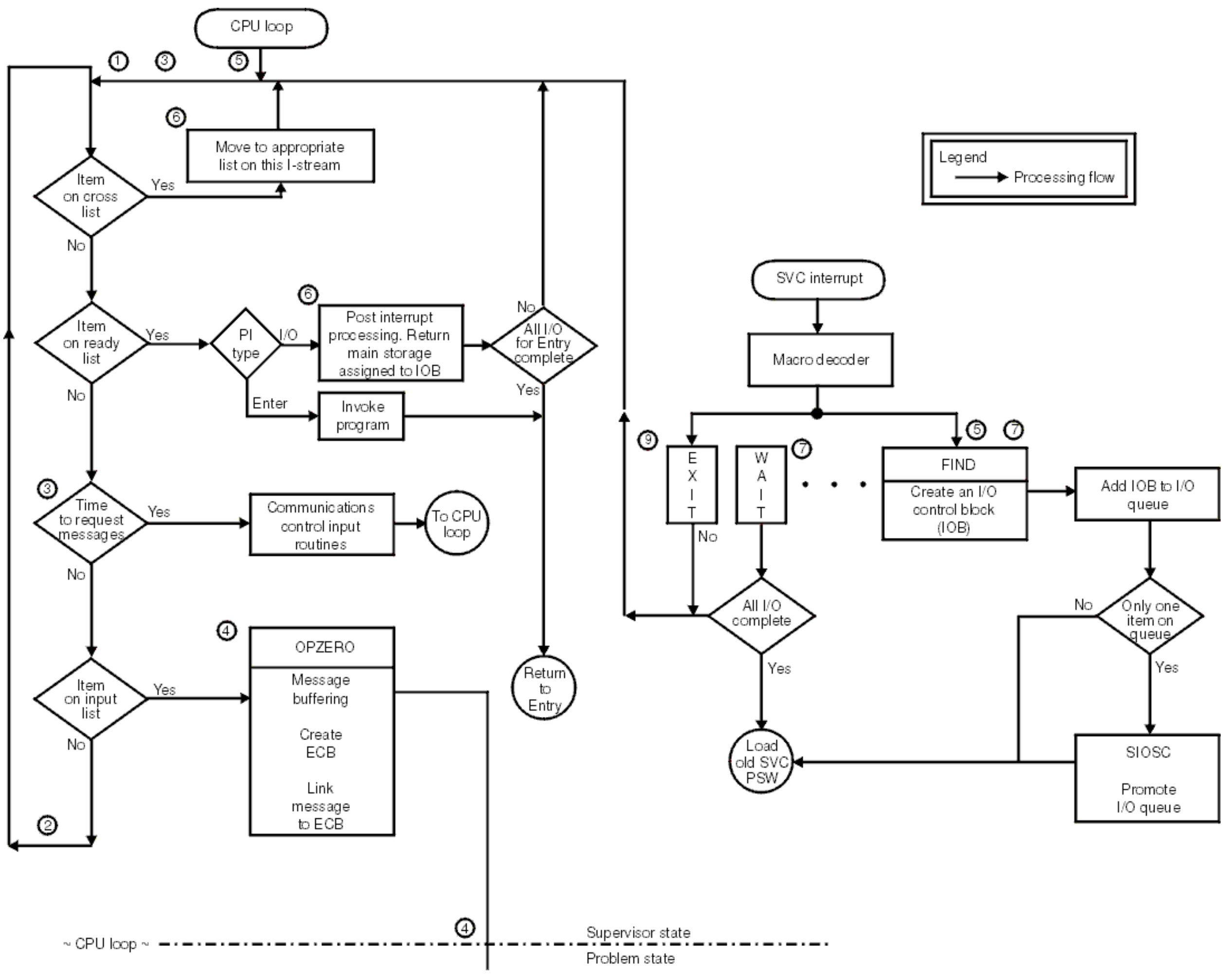
Transaction:

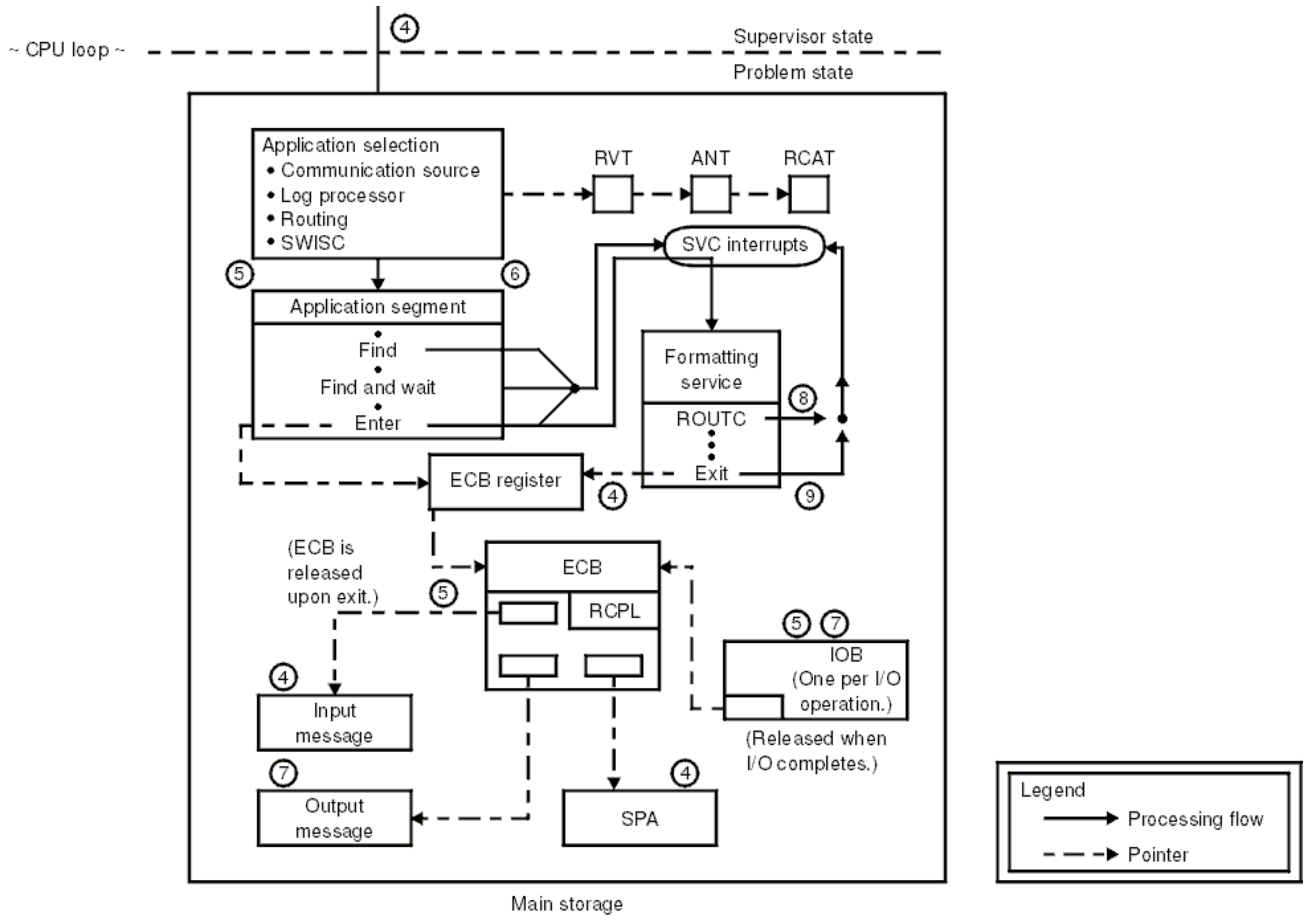
- Collection of entry makes a transaction. For example creating a PNR or Taking money from ATM Machine.

- While creating a PNR, entries (input messages) are used to store the Name, Phone number, Board point, off point, Agent name, time limit, address and when ET (end transaction) is used at that time all passenger related data are stored in Database.

2.3 Message flow through zTPF system

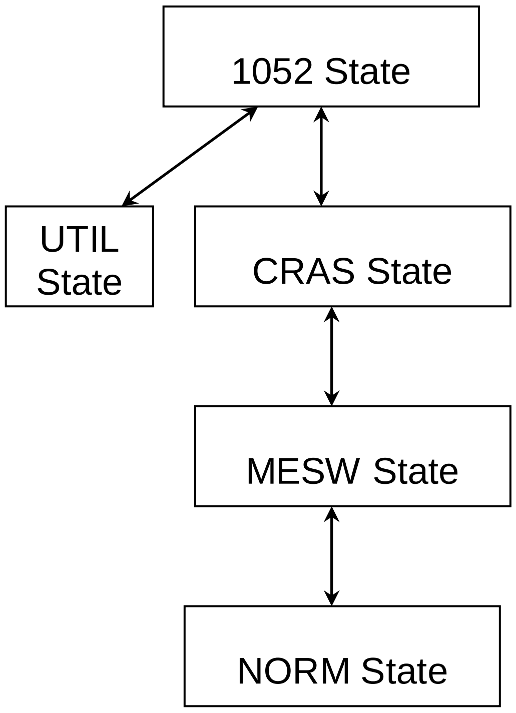
- Following diagrams illustrates the flow of message through a ZTPF system:





- Step 1 – The system is initialized.
- Step 2 – CPU loop checks for work on the cross, time dispatch, ready, and input lists.
- Step 3 – Input message arrives.
- Step 4 – zTPF creates an ECB and selects the application.
    a. Message Preprocessing.
    b. OPZERO Creates and initializes the ECB.
    c. COMM source invokes the application.
    d. Application is selected.
- Step 5 – Fetch application program from file.
- Step 6 – Start Program execution.
- Step 7 – Run Application.
- Step 8 – Send response to message source.
- Step 9 – Release system resources and clean-up.

2.4 System States



- **1052 State** - This is the lowest state, where most of the system services are not available. Only two functions are available at 1052 state. The 3270 Prime CRAS operations, Key-pointing.

- **UTIL State** - The UTIL state can be entered only from the 1052 state. The state is similar to 1052 but the extra services available are functional messages from CRAS, real time clock, time-initiated entries, tape and disk interrupt processing. This state is of negligible significance to Application Programming.

- **CRAS State** - The CRAS state is the state entered next. Apart from the functions mentioned above, CRAS entries are permitted, GFS(Get File Storage) is active. This state is of negligible significance to Application Programming.

- **MESW State** - The MESW state is similar to the CRAS state. But the additional features are Non-SDLC (Synchronous Data Link Control) lines are active and only message switching entries are permitted. All other messages are logged for later processing. This state is of negligible significance to Application Programming.

- **NORM State** - At NORM state all the functions are active and application processing is started. The TAS is initiated, catastrophic error recovery is attempted if needed, and NCP polling begins.

2.5 zTPF Programming Conventions

The zTPF system has established certain programming conventions to achieve high performance and ease program maintenance. It is important that the application conform to these conventions. The zTPF system enforces some of the conventions. However, only good programming discipline can maintain others.

Below is a summary listing of assembler language programming conventions:

- Application programs should not execute privileged instructions, nor explicitly issue the supervisor call (SVC) instruction.
- The general registers 0 through 7 are reserved for use by application programs. Their contents are saved across all control program macros. Registers 14 and 15 may also be used by application programs, but are often used by the control program to pass information between programs. The contents are not guaranteed across macro calls unless otherwise indicated.
- General register 8 always contains the base address of the active application program. It must not be altered by application programs.
- General register 9 always contains the address of the current active entry control block. It must not be altered by application programs.
- General register 10 can be used as a scratch register (the contents are not guaranteed across macro calls).
- General registers 11 and 12 are reserved for control program use. Their contents must not be modified by an application program. Register R11 contains a value of X'1000' and register R12 contains a value of X'2000'.
- General register 13 can be used as a scratch register (the contents are not guaranteed across macro calls).
- General registers 10 to 13 can be used by application programs that use the *extended register save* function either by coding the EREGSAVE parameter on the BEGIN macro or by using the DEFBC macro. By using the extended register save function, the contents of R10–R13 are saved across all general macro calls.
- Programs that run on a zTPF system can access ECB heap in 31-bit or 64-bit addressing mode.
- C shared object (CSO) types include:
    - Single entry point with a maximum of one main function.
    - Single entry point without a main function, which can include program libraries, or non-program libraries, or a combination of both.
    - Non-program libraries.

3. Entry Control Block

3.1 ECB Overview

- The entry control block, usually called the ECB, is the cornerstone of the application program re-entrant structure in the zTPF system.
- Use of the ECB allows one program to service multiple Entries because an ECB is private to an Entry.
- It is analogous to the "Process" in other operating systems.
- The system has in the ECB, control data that relates to an entry.
- The application puts in it business data. The application uses this ECB as working storage during the life time of the entry.
- Programs store data in the ECB to achieve reentrancy. Storing data in the program is not allowed in TPF.

Violates reentrancy	Does not VIOLATE reentrancy
`$IS$  CSECT`	`$IS$  CSECT`
`      LHI   R1,1`	`      LHI   R1,1`
`      AHI   R1,10(R1)`	`      AHI   R1,10(R1)`
`      STH   R1,LABEL`	`      STH   R1,EBW000`
`          .`	`          .`
`          .`	`          .`
`          .`	`          .`
`LABEL DS    H`	`      LTORG`
`          .`	`      FINIS ABCD`
`          .`	`      END`
`          .`	
`      LTORG`	
`      FINIS ABCD`	
`      END`	

- The ECB is of size 12 K. The size is not variable.
- The ECB is divided into three pages each of size 4K.
- The General Purpose Register R9 points to the ECB.
- The system programs, during ECB creation load the ECB address into the register.

3.2 Format of the ECB

- The ECB is 12 KB long and is referred to in three 4-KB page segments.

- ECB page 1 is primarily used by applications and is the link between the zTPF system and the application program (ECB-controlled program). It contains:

  - Fixed application work areas, which are used by the application program

    This is work space for use by application programs in any way it is required. The work areas are unformatted and of fixed sizes. However, if an application requires additional work area beyond what is provided by the ECB, it can be requested from the zTPF system.
  - Fixed system work areas
    - Resource interface area, which is used by both the application program and the control program

      This area contains the addresses of file records and working storage blocks that the application program has requested from the zTPF system.

    - Entry management area, which is used by the control program. This area contains status information and other data required for managing an entry.

  - User-definable area, which can be used by the application program. The space is for use by application programs in any way it is required.

System Work Area - 1 (Chain word, PIBA)
Application Work Area (EBW)
Data Levels
System Work Area - 2
Register Saver Area - User (CE1URA-UR7)
Application Work Area - Extended (EBX)
System Work Area - 3
E-type Macro Work Area
System Work Area - 4
Variable length User Work Area (2752 bytes)

- ECB page 2 is used primarily by the control program.
- It is used to hold data related to the Entry that might impact system availability if damaged by the application program.
- Such as control information that is used to manage the ECB and working storage blocks.
- ECB page 2 also contains a field that points to the **resource limit table** (RLT), which contains the two levels of limits for each resource that is monitored.
- There is also a user-definable area in ECB page 2. The difference between this user-definable area and the one in ECB page 1 is that this one offers an additional level of protection because its storage key is not the same as the storage key for the entry.
- Protection is achieved because the storage protection key must be explicitly changed in order to update this area.

Working Storage management area/ECB addresses
Block Check Mode Flag area
Heap Management area
E-type loader related data area
ISO-C support area
Resource Control related data area
Suspended list management/Time Slicing work area
Other fields

- ECB page 3 contains a combination of application and system work areas and control fields.
- There also is a field in ECB page 3 that points to the resource counter table (RCT), which contains the counter for each monitored resource.

ECB Macro trace table (2664 bytes) 66 entries of size 40 bytes
Program nesting levels (980 bytes) 35 entries of size 28 bytes
Detached Block Table (360) 10 entries of size 36 bytes
Other system work areas (PAT address of current program etc)

3.3 Data levels

- Each entry has the capability of referencing 16 data blocks concurrently in the ECB.
- The ECB maintains these references in sixteen data levels numbered hexa decimally from 0 to F (often referred to as D0–DF).
- A data level is simply a series of double word reference/control fields for data blocks that can be used however the application requirements dictate.
- There are three sets of double word references for each data level. They are, in order of their physical position in the ECB:
  - File address reference words (FARW)
  - Core block reference words (CBRW)
  - File address extension words (FAXW).
- The **FARW** is used to record the file address and control data related to file I/O for each data level.
- There may be a main storage block on a given level without any file activity, in which case the FARW will not be used.
- When there is file activity the FARW will be used for the record identification (2 bytes), record code check (1 byte), and symbolic file address (4 bytes); 1 byte is unused.
- In the below example, the ‘y’ refers to the Data level number which can be any hexa decimal value 0 – F.

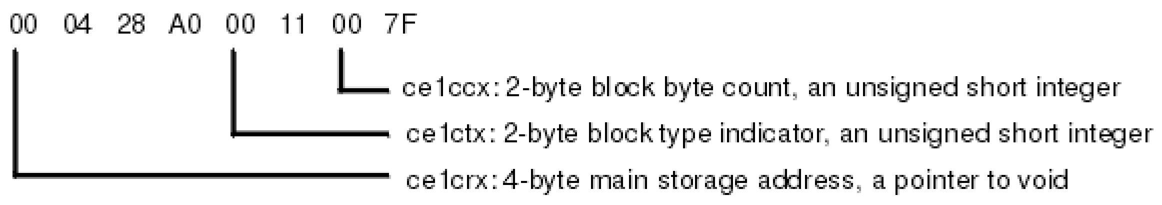
RECORD ID	RCC	NOT USED	FILE ADDRESS

0 2 3 4 7

- The **CBRW** is used to store the main storage address and control information about main storage blocks used by the entry.
- zTPF programs format the CBRW whenever a main storage block is attached to or detached from the ECB.
- The format consists of the main storage address (4 bytes), the block type indicator (2 bytes), and the block byte count (2 bytes).

- The main storage block address has the same label as the CBRW. The block type indicator and the block byte count have their own labels.

- In the following example, x is the data level number, a hexadecimal digit from 0 to F.


00 04 28 A0 00 11 00 7F
ce1ccx: 2-byte block byte count, an unsigned short integer
ce1ctx: 2-byte block type indicator, an unsigned short integer
ce1crx: 4-byte main storage address, a pointer to void

- The block type indicator (CE1CTX) specifies the size of the block attached to the ECB at level X.

0001	No block attached	
0011	127-byte block attached	(L0)
0021	381-byte block attached	(L1)
0031	1055-byte block attached	(L2)
0051	4095-byte block attached	(L4)

- The block byte count (ce1ccx) specifies the number of bytes in the block that the program can access.

- It is important to note that only the block type indicator accurately reflects the status of the level at any given time.

- If ce1ctx contains X'0001', then the data in the rest of the CBRW for level x is not meaningful.

- The zTPF system does not initialize ce1crx and ce1ccx after a block is released.

- Only if ce1ctx contains X'0011', X'0021', X'0031' or X'0051' is a currently valid main storage address in the level x CBRW.

- The **File address extension word (FAXW)** is used to pass information between zTPF online systems and z/OS, using DASD files called *general data sets*.

3.4 Application Work areas

- There are two application work areas, each 112 bytes long, consisting of a 104-byte scratch area followed by an 8-byte bit-switch area.

- The first work area begins at ebw000. Each byte of its scratch area is named sequentially beginning with ebw000 and ending with ebw103.

- The second work area begins at ebx000. Each byte of its scratch area is named sequentially beginning withebx000 and ending with ebx103.

- The last 8 bytes in each of these work areas are intended for use as program switches.

- The program switch names and their standard usage conventions are as follows:

    - **ebsw01–ebsw03** – Inter-program switch to specify various conditions among programs
    - **ebrs01** - Used to pass error information among program segments
    - **ebcm01–ebcm03** - Intra-program switch to specify various conditions within programs
    - **eber01** - Used to pass error information within program segments.
    - **ebxsw1**– **ebxsw7 –** General Indicator bytes.

- OPZERO initializes the switches—that is, the last eight elements—in both work areas, to zero when it creates the ECB.

- The first 104 elements of the work areas are not initialized, so their contents are unpredictable.

- Two fields in the ECB are used to indicate unusual conditions associated with I/O requests:
    - CE1SUG is a 1-byte gross or summary indicator of all unusual conditions occurring on any level.
    - CE1SUD is a series of 16 indicators--1 byte for each level.

- The application should never clear or modify CE1SUG, and check CE1SUD immediately after an I/O request has ended.

- Q1. How ECB helps in ensuring the reentrancy?
- Q2. Can we define storage in program and how?
- Q3. Defining a DSECT in program will violates the reentrancy concepts, explain?
- Q4. Which field identifies whether core block is held or not?

4. Working Storage Management

4.1 zTPF Main Memory Layout

- The organization of main storage for program and data management is the key to the design of a high-performance system such as zTPF.
- Storage protection is used for fixed storage to prevent an application from illegally modifying programs and critical data records.
- Part of main storage contains programs and data that are common to all entries and that always remaining main storage.
- This portion is called **fixed storage** in the z/TPF system.
- Another portion, called **working storage**, is allocated to entries as required.
- Programs and data are transferred from file storage to working storage as required.
- Fixed storage consists of the following main storage areas:
    - Control program
    - Core resident program area (CRPA)
    - Application format-1 global area
    - Control program records and table area.
- Below is a detailed illustration of the main memory layout in zTPF which describes the different components and protect keys.

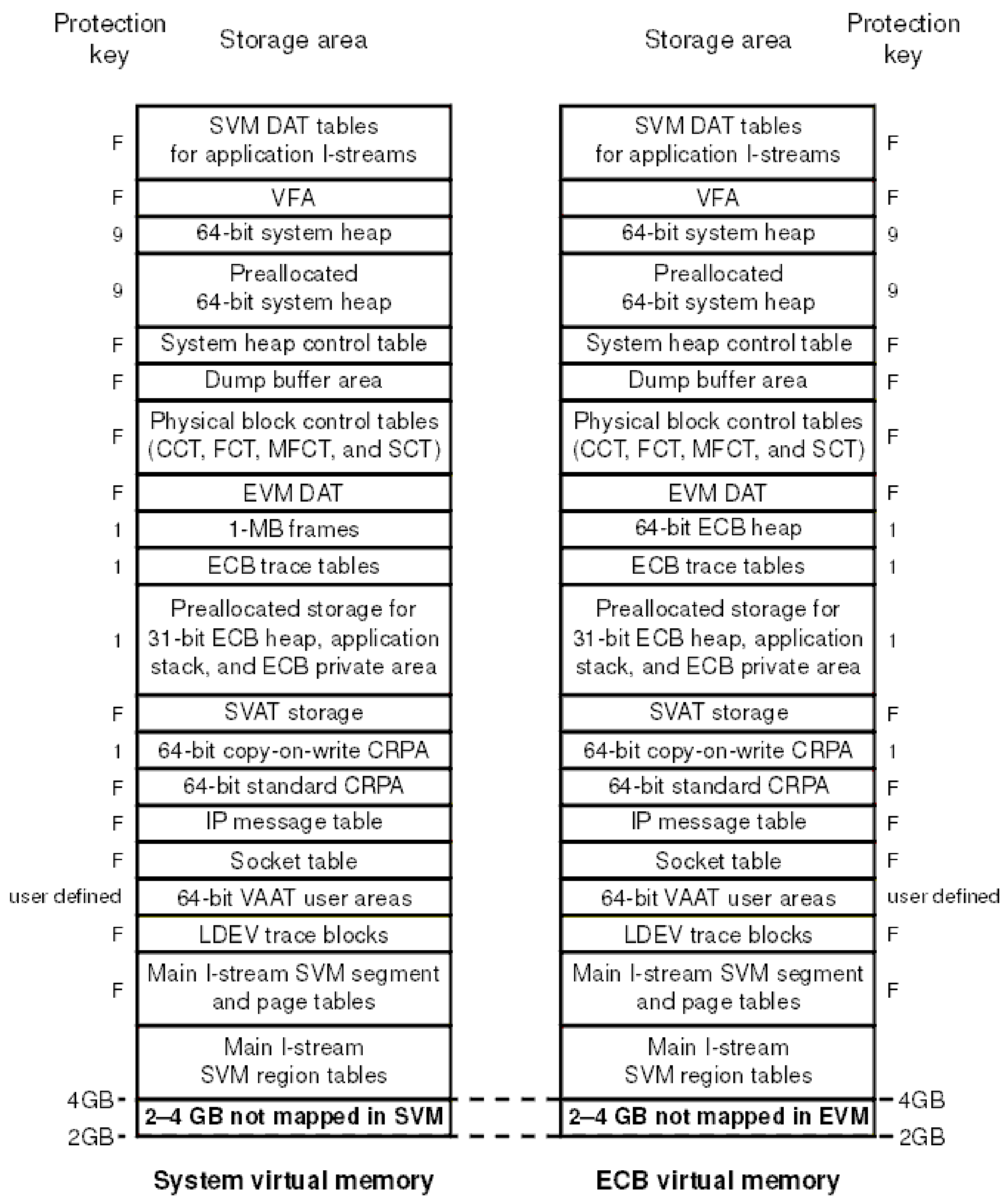

Protection key
Storage area
Storage area
Protection key
F | SVM DAT tables for application I-streams | F
F | VFA | F
9 | 64-bit system heap | 9
9 | Preallocated 64-bit system heap | 9
F | System heap control table | F
F | Dump buffer area | F
F | Physical block control tables (CCT, FCT, MFCT, and SCT) | F
F | EVM DAT | F
1 | 1-MB frames
64-bit ECB heap | 1
1 | ECB trace tables | 1
1 | Preallocated storage for 31-bit ECB heap, application stack, and ECB private area | 1
F | SVAT storage | F
1 | 64-bit copy-on-write CRPA | 1
F | 64-bit standard CRPA | F
F | IP message table | F
F | Socket table | F
user defined | 64-bit VAAT user areas | user defined
F | LDEV trace blocks | F
F | Main I-stream SVM segment and page tables | F
Main I-stream SVM region tables
4GB
2–4 GB not mapped in SVM
2–4 GB not mapped in EVM
2GB
System virtual memory
ECB virtual memory

Figure 1 : Main memory Layout Part 1

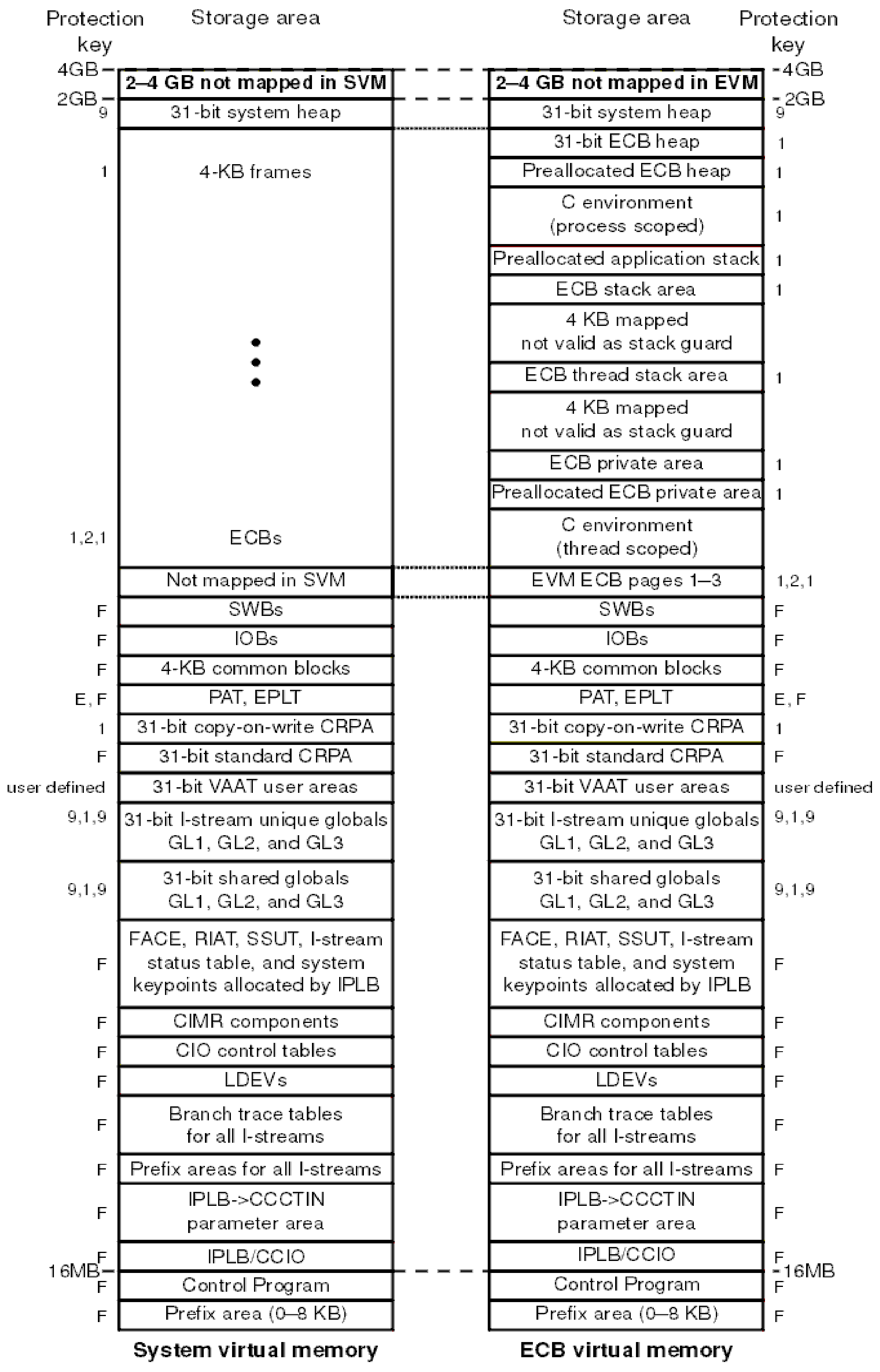

Protection key
Storage area
Storage area
Protection key
4GB
2GB
2–4 GB not mapped in SVM
2–4 GB not mapped in EVM
4GB
2GB
9
31-bit system heap
31-bit system heap
9
1
4-KB frames
31-bit ECB heap
1
Preallocated ECB heap
1
C environment (process scoped)
1
Preallocated application stack
1
ECB stack area
1
4 KB mapped not valid as stack guard
ECB thread stack area
1
4 KB mapped not valid as stack guard
ECB private area
1
Preallocated ECB private area
1
1,2,1
ECBs
C environment (thread scoped)
Not mapped in SVM
EVM ECB pages 1–3
1,2,1
F
SWBs
SWBs
F
F
IOBs
IOBs
F
F
4-KB common blocks
4-KB common blocks
F
E, F
PAT, EPLT
PAT, EPLT
E, F
1
31-bit copy-on-write CRPA
31-bit copy-on-write CRPA
1
F
31-bit standard CRPA
31-bit standard CRPA
F
user defined
31-bit VAAT user areas
31-bit VAAT user areas
user defined
9,1,9
31-bit I-stream unique globals GL1, GL2, and GL3
31-bit I-stream unique globals GL1, GL2, and GL3
9,1,9
9,1,9
31-bit shared globals GL1, GL2, and GL3
31-bit shared globals GL1, GL2, and GL3
9,1,9
F
FACE, RIAT, SSUT, I-stream status table, and system keypoints allocated by IPLB
FACE, RIAT, SSUT, I-stream status table, and system keypoints allocated by IPLB
F
F
CIMR components
CIMR components
F
F
CIO control tables
CIO control tables
F
F
LDEVs
LDEVs
F
F
Branch trace tables for all I-streams
Branch trace tables for all I-streams
F
F
Prefix areas for all I-streams
Prefix areas for all I-streams
F
F
IPLB->CCCTIN parameter area
IPLB->CCCTIN parameter area
F
F
IPLB/CCIO
IPLB/CCIO
F
16MB
16MB
F
Control Program
Control Program
F
F
Prefix area (0–8 KB)
Prefix area (0–8 KB)
F
System virtual memory
ECB virtual memory

Figure 1 : Main memory Layout Part 2

4.2 Memory Security

- The zTPF system uses virtual addressing, which restricts ECBs from accessing other ECBs.
- When a New ECB is created, it has access only to its own storage; it cannot see or modify another ECB's storage.

- Below described are three key concepts of Memory security in zTPF.

    - Storage Protection
    - Block check Mode
    - Heap check Mode

4.2.1 Storage Protection

The z/TPF system makes use of three kinds of protection to protect main storage from destruction or misuse.

i) Key-controlled protection

This minimizes unauthorized fetches and stores of data. When key-controlled protection applies to a storage access, a program can store data only when the storage key matches the access key and a program can fetch data only when the keys match or when the fetch-protection bit of the storage key is zero.A storage key is associated with each 4 KB block of storage. These storage keys are not part of addressable storage. The access key is usually the program status word (PSW) key and occupies bit positions 8 to 11 of the current PSW.

ii) Dynamic address translation (DAT) protection

This provides protection against unauthorized storing. DAT protection uses the DAT-protection bit in each page table entry and segment table, and, when the enhanced-DAT facility is installed, in each region table entry to control access to virtual storage.

iii) Low-address protection

This protects the first 512 bytes of pages 0 and 1 from destruction. These areas are critical during interrupt processing. One difference between low-address and key-controlled protection is that low-address protection does not prevent corruption by the channel subsystem, but key-controlled protection does.

4.2.2 Block Check Mode

Block check mode is a debugging tool that flags certain coding errors, such as writing beyond the end of a block, passing blocks chained to other blocks, and using storage that has already been released. When block check mode is on:

- ECBs run in single block mode. ***Single block mode*** dispenses a single block in each frame. The block is located in the last block slot available in the frame. During ECB initialization, the pre-allocated 4-KB blocks will not be available if the ECB is running in single block mode. Single block mode is suspended automatically for an individual ECB if the system cannot find enough available blocks in the ECB private area. The number of available blocks that are required equals the maximum private area size, in megabytes, multiplied by 8.

- For 128-byte block and 381-byte block type assigned block size is 381 bytes.

- ***Release block processing*** disconnects a frame from an ECB if the block being released is the only block in the frame. Subsequent references to the block will result in a page fault because the address is no longer valid in the address space for the ECB.

- *ECB* ***exit processing*** interrogates each frame it disconnects from the ECB to look for lost blocks. If a block is found to be flagged **in use**, then a 000749 system error will occur to indicate that a missing block has been found.

4.2.3 Heap Check Mode

- Heap check mode, which is similar to the existing block check mode, is a debugging tool that flags certain coding errors related to the use of ECB heap.

- It is to help the programmer determine when an application is changing or accessing storage beyond its allocated heap buffer or accessing an ECB heap buffer after it was released.

4.3 Memory management macros

GETCC – Get a core block

Assigns a core block to ECB based on parameters provided.

Syntax:

Assembler:

GETCC Dn,Ln,FILL=xx,SIZE=350

Dn Specifies the data level (D0-DF) on which the core block has to be assigned.

Ln Specifies the block type needed. The block size is implicit. The values may be L0, L1, L2 or L4.

FILL Specified if the core block needs to be initialized with a specific hexadecimal value (xx) when assigned to the ECB. The contents of the core block are unpredictable when the FILL parameter is not used.

SIZE Size can be given as in bytes or a register in the form (reg) containing the block size. An example of a value for block size might be SIZE=350. The largest block size supported is 4095. It should be expressed either by using the hexadecimal type of specification or by the decimal numeric constant.

Note: The type (L0, L1, L2 and L4) and SIZE parameter are mutually exclusive.

C:

void *getcc(enum t_lvl level, enum t_getfmt format, spec1, spec2);

level One of 16 possible values representing a valid ECB data level from the enumeration type *t_lvl*, expressed as D*x*, where *x* represents the hexadecimal number of the level (0-F). This parameter represents an available core block reference word (CBRW) on which the requested working storage block will be placed.

format or this parameter you must specify at least one term of enumeration type *t_getfmt*, defined in *tpfapi.h*, to determine how storage is allocated. Additionally, you can code the terms GETCC_FILL and GETCC_COMMON or GETCC_PROTECTED. If you code more than one term for *format*, you must separate terms with a plus sign (+).

spec1 The *getcc* function always requires a third parameter depending on what term is coded for *format*.

If GETCC_TYPE is coded, *spec1* must be a logical block type.

If GETCC_SIZE is coded, *spec1* must be an integer indicating the number of bytes of storage you need.

If one of GETCC_ATTR0 through GETCC_ATTR9, GETCC_PRIME, or GETCC_OVERFLOW is coded, *spec1* must be a 2-character record ID in double quotes.

spec2	If GETCC_FILL is coded, you must code a fourth parameter specifying the hex value to which you want storage initialized.

Other variants in C:

void	*getcc(enum t_lvl level, enum t_getfmt format, spec1);

Entry Conditions:

- The CBRW of the Level specified should not have a core block attached.

Return Conditions:

Assembler:

- The address, Logical size and block type are placed in the respective CBRW in the ECB.
- Register R14 contains address of the core block, Register R15 contents are unknown.
- All other registers are preserved over this macro call.

Note: If more than 4095 bytes require then go for Heap area (CALOC or MALOC) instead of core block (GETCC). Also need to take care whether it is free or not etc..etc..

C:

- Pointer to the newly obtained working storage block.

Exception:

CTL 6	Level Specified already has a block attached.

Programming Consideration:

- Using FILL parameter consumes more CPU resources.

Example:

Assembler:

```
GETCC  D4, L4, FILL=00  Gets a 4K block on Level 4 initialized with hex zeros.
GETCC  D4, L4, FILL=40  Gets a 4K block on Level 4 initialized with blanks.
GETCC  D1,L1                Get work area of 128 bytes, No initialization
GETCC  D2,SIZE=350,FILL=40      Get the work area of 350 bytes, initialized with blanks
```

C:

```
char *pt1;

pt1=(char*)getcc(D4,
     (enumt_getfmt)(GETCC_TYPE+GETCC_FILL),L4,0x00);
```
Gets a 4K block on Level 4 initialized with hex zeros.

```
pt1=(char*)getcc(D4,
     (enumt_getfmt)(GETCC_TYPE+GETCC_FILL),L4,'');
```
Gets a 4K block on Level 4 initialized with blanks.

```
pt1=(char*)getcc(D1,GETCC_TYPE,L1);
```
Gets work area of 128 bytes on Level 1, No initialization.

```
Pt1=(char*)getcc(D3,
     (enumt_getfmt)(GETCC_SIZE+GETCC_FILL),350,'');
```
Gets a 350 bytes block on Level 3 initialized with blanks.

RELCC – Release a core block

Release a core block attached to the specified Data level in the ECB.

Syntax:

Assembler:

RELCC Dn

Dn	Specifies the data level (D0-DF) from which the core block has to be released.

C: ***void relcc(enum t_lvl level);***

Level	One of 16 possible values representing a valid entry control block (ECB) data level from enumeration type *t_lvl*, expressed as D*x*, where *x* represents the hexadecimal number of the level (0–F). This parameter identifies the core block reference word (CBRW) containing the address of the working storage block to be returned to the system.

Other variants in C: None

Entry Conditions:

- The CBRW of the Level specified should have a core block attached.

Return Conditions:

Assembler:

- The CBRW specified Level is updated to indicate absence of the core block.
- Register R14 and R15 contents are unknown.
- All other registers are preserved over this macro call.

C:

Void

Exception:

CTL- 7	Level Specified does not have a block attached.
CTL– B	The block type in the specified level is invalid
CTL– 6D9	The user attempts to release the block twice.
CTL-6DA	The address in CBRW at the level is invalid or not a block boundary.

Programming Consideration: Only one storage block is released by one RELCC macro call. If multiple blocks need to release then call RELCC macro multiple times.

Example:

Assembler:

```
RELCC   D7
```

Releases core block on Level 7

C:

```
relcc(D7)
```

Releases core block on Level 7

LEVTA – Level Test

Determine whether a core block is being held at a specified data level in the ECB. It can be supplied with two symbolic addresses, one for return of control is block is present and one for return if block is not present. At least one of these addresses must be supplied.

Syntax:

Assembler:

LEVTA LEVEL=n, INUSE=labelyes, NOTUSED=labelno

LEVEL	The data level (0-F) to be tested for a core block presence.
INUSE	The location to branch to if a core block is present in the specified level.
NOTUSED	The location to branch to if a core block is not present in the specified level.

C:

int levtest(enum t_lvl level);

Level	One of 16 possible values representing a valid entry control block (ECB) data level from the enumeration type *t_lvl*, expressed as D*x*, where *x* represents the hexadecimal number of the level (0–F). This parameter identifies the CBRW to be tested for the presence of a working storage block.

Other variants in C: None

Entry Conditions:

- Either INUSE, NOTUSED or both must be specified.

Return Conditions:

Assembler:

- If only the INUSE parameter is used, an ECB data level or DECB holding a core block will lead to a branch to the location specified by INUSE. If no core block is held at that ECB data level or DECB, control will be returned to the next sequential instruction (NSI).
- If only the NOTUSED parameter is used and no core block is held at the tested ECB data level or DECB, control will be returned to the symbolic location specified by the NOTUSED parameter. If a core block is held at the tested ECB data level or DECB, control will be returned to the NSI.
- If both NOTUSED and INUSE are used, control will be returned to the symbolic location specified by the parameter that satisfies the condition of the ECB data level or DECB tested. In this case code below LEVTA is dead code if symbolic location specified for INUSE or NOTUSED is not NSI.

C:

- An integer value representing the size of the working storage block in bytes if the ECB data level or DECB is occupied, or zero if the ECB data level or DECB is unoccupied.

Exception: None

Programming Consideration: None

Example:

Assembler:

```
LEVTA LEVEL=8,INUSE=DETAC8,NOTUSED=ATTAC8
```

Control branches to label DETAC8 If core block present else Branches to ATTAC8.

C:

```
if(levtest(D6))
  {
    detac(D6);
  }
```

Detac level 6 if level 6 already present.

CRUSA – Test and Release Core Block

CRUSA is used to test and release core block(s) that is present in the data levels specified.

Syntax:

Assembler:

CRUSA Sm=n,TEST=n,PARAM=label1

Sm=n	**m** is an incremental parameter that starts with 0,1,2,... in order. It is NOT a Data level. **n** specifies the data levels to test for core blocks being held. Any core blocks found are released. At least one level (0-F) must be specified; if more than one level is specified. They can be in any order.
TEST	n specifies the data level on which the check has to be performed. An absolute value 0-F is expected. This parameter is used with the **PARAM** parameter to test and branch if core block not held.
PARAM	Specifies the address to be branched to, if a block is not present. If block is present the control is transferred to the NSI.

C:

void crusa(int count, enum t_lvl level,…);

count	An integer containing the number of ECB data level or DECB parameters included in the parameter list.
level	One of 16 possible values representing a valid data level from the enumeration type t_lvl, expressed as Dx, where x represents the hexadecimal number of the level (0–F). The working storage block on this data level is the block to be detached.

Other variants in C:

void	crusa(int count, TPF_DECB *decb,…);

Entry Conditions:

- The specified data level may or may not have a core block attached.

Return Conditions:

Assembler:

- All released core block reference words are initialized to indicate the absence of core block.
- Register R14, R15 contents are unknown and other registers are preserved across this macro call.

C:

- The CBRW has been modified to indicate that no block is held.

Exception: None

Programming Consideration:

- CRUSA is a logical combination of LEVTA and RELCC.

Example:

Assembler:

```
CRUSA S0=3,S1=4        Data level 3 & 4 are tested for attached block and released.
CRUSA TEST=1,PARAM=LABEL test data level 1 and branch to
                         LABEL is it is present else NSI.
CRUSA S0=3,S1=4,TEST=1,PARAM=LABEL this is combination of
                                   above two examples.
```

C:

```
crusa(2,D3,D4);    Data level 3 & 4 are tested for attached block and released
```

FLIPC – Interchange status of Data levels.

Interchanges the data contained in the ECB control fields associated with to data levels, which includes CBRW, FARW, FAXW and detail error indicator fields. Mostly this macro is used when some program or utility expects data on particular level (example D3) and current program has data on D5 then FLIPC D3, D5 and call other program or utility.

Syntax:

Assembler:

FLIPC Dx, Dy

Dx, Dy	Specifies the data levels (D0-DF) of which the fields should be interchanged.

C:

void flipc(enum t_lvl level1, enum t_lvl level2);

level1	One of 16 possible values representing a valid data level from the enumeration type *t_lvl*, expressed as D*x*, where *x* represents the hexadecimal number of the level (0–F). This is the first data level that participates in the status interchange.
level2	One of 16 possible values representing a valid data level from the enumeration type *t_lvl*, expressed as D*x*, where *x* represents the hexadecimal number of the level (0–F). This is the second data level that participates in the status interchange.

Other variants in C: None

Entry Conditions:

- The specified data levels should not have any I/O operation in progress.

Return Conditions:

Assembler:

- Register R14, R15 contents are unknown and other registers are preserved across this macro call.

C:

void

Exception: None

Programming Consideration:

- It is not mandatory that the data levels should hold a core block since it is just the fields associated that get swapped and not the block itself.
- If the same level is specified for both data levels, the macro expansion will result in “NOP”

instruction.

Example:

Assembler:

```
   FLIPC D7,D3
```

Flips contents of Data levels 7 and 3.

C:

```
   flipc(D7,D3);
```

Flips contents of Data levels 7 and 3.

DETAC – Detach a Working Storage Block

DETAC detaches a core block from the specified data level, allowing the Data level to be reused. The detached core block is saved and can be reattached by using the ATTAC macro.

Syntax:

Assembler:

DETAC Dn,CHECK=NO or YES

Dn	Specifies the data levels (D0-DF) from which the core block is to be detached.
CHECK	**YES** – The service routine will issue a system error upon finding no core block attached to the data level. The Default value for CHECK parameter is YES.
	NO – The system will make the data level reusable, without validating for a block to be present.

C:

void detac_ext(enum t_lvl level, int ext);

Level	One of 16 possible values representing a valid data level from the enumeration type t_lvl, expressed as Dx, where x represents the hexadecimal number of the level (0–F). The working storage block on this data level is the block to be detached.
Ext	**DETAC_CHECK** Indicates that the ECB data level or DECB being detached should be checked to ensure there is a block to be detached. **DETAC_NOCHECK** Indicates that no check will be made to determine if there is a block to be detached. If DETAC_NOCHECK is specified, the service routine will make the ECB data level reusable without validating that a block is held.

Other variants in C:

Void	detac enum t_lvl level);
Void	detac_id(enum t_lvl level);

Entry Conditions:

- When CHECK=YES in case of Assembler or DETAC_CHECK in case of C is specified, the data level should be holding a core block.

Return Conditions:

Assembler:

- The specified data level will no longer hold a core block.

- Register R14, R15 contents are unknown and other registers are preserved across this macro call.

C:

- The CBRW has been modified to indicate that no block is held.

Exception:

CTL- D2	If core block not present in specified data level or Number of DETACs in ECB exceeds 255.

Programming Consideration:

- A DETAC counter is maintained in the ECB for each data level with a maximum limit of 255 DETACs per data level.
- When DETAC count is exceeded, the service routine will issue a system error.

Example:

Assembler:

`DETAC D2`	*Detaches core block from data level 2*.
`DETAC D3,CHECK=NO`	*Detaches core block (if any) from data level 3*.

C:

`detac(D6);`	*Detaches core block from data level 6*.
`detac_ext(D6,DETAC_NOCHECK);`	*Detaches core block from data level 6, if any*

ATTAC – Reattach a Working Storage Block

Attaches a core block that was detached using DETAC to the specified data level.

Syntax:

Assembler:

ATTAC Dn

Dn	Specifies the data levels (D0-DF) from which the core block is to be reattached.

C:

void ****attac(enum t_lvl level);***

Level	One of 16 possible values representing a valid data level from the enumeration type t_lvl, expressed as Dx, where x represents the hexadecimal number of the level (0–F). The working storage block on this core block reference word (CBRW) level is the block to be attached.

Other variants in C:

void	*attac_ext(enum t_lvl level, int ext);
void	*attac_id(enum t_lvl level);

Entry Conditions:

- A core block must have been previously detached from the specified level.
- The specified data level should NOT contain a core block.

Return Conditions:

Assembler:

- The specified data level will be holding a core block, with respective CBRW, FARW and FAXW contents from the previous DETAC.
- Register R14, R15 contents are unknown and other registers are preserved across this macro call.

C:

- The pointer of type `void` representing the address of the start of the newly attached block.

Exception:

CTL- D1	If core block is present in specified data level or NO DETAC was done on the specified level previously.

Programming Consideration:

- ATTAC will always attach the block most recently detached from the specified data level.

Example:

Assembler:

```
ATTAC D2
```
Reattaches core block to data level 2.

C:

```
attac(D6);
```
Reattaches core block to data level 6.

MALOC – Reserve a Storage Block

Obtains a variable-sized, double word aligned storage block from Heap.

Syntax:

Assembler:

MALOC SIZE=Rn

SIZE	**Rn** – General purpose register (R0-R7, R14 or R15) containing the number of contiguous bytes of storage to be allocated.

C:

void *malloc64(size_t size);

size	The length, in bytes, of the requested storage area.

Other variants in C: None

Entry Conditions:

- The specified register contains the non-zero number of bytes requested.

Return Conditions:

Assembler:

- The specified register contains the beginning address of the allocated heap.
- Register R14, R15 contents are unknown and other registers are preserved across this macro call, expect for the register specified in SIZE parameter.

C:

- Returns a pointer to the area of 64-bit ECB heap that was reserved.

Exception:

CTL- 7B	Heap storage exhausted.

Programming Consideration:

- Any valid register for an ECB-controlled program (R0-R7, R14 or R15) can be used for size parameter.
- Allocated memory has Garbage value, programmer responsibility to initialize the block.

Example:

Assembler:

```
LA R4,4095
MALOC SIZE=R4
```

`LA R4,4095`	*Allocates 4K bytes of heap storage*
`MALOC SIZE=R4`	*Returns the address in register R4*

C:

```
char *pt1;

pt1=(char*)malloc64(500));
```

Allocates 500 bytes of memory and returns the start address to the pointer pt1.

CALOC – Reserve and Initialize a Storage Block

CALOC obtains a variable-sized, double word aligned storage block from heap and initializes it to zeros.

Syntax:

Assembler:

CALOC COUNT=Rx,ESIZE=Ry

COUNT	**Rx** – General purpose register (R0-R7, R14 or R15) containing the number of blocks of storage to be allocated.
ESIZE	**Ry** – General purpose register (R0-R7, R14 or R15) containing the size of each block to be allocated..

C:

void *calloc64(size_t num,size_t size);

num	The number of elements to be allocated.
size	The length, in bytes, of each element.

Other variants in C: None

Entry Conditions:

- The specified registers contain the non-zero number of blocks and size requested.

Return Conditions:

Assembler:

- The specified register in COUNT contains the beginning address of the allocated heap.
- Storage allocated is initialized to zeros.
- Content of register specified in ESIZE is unaltered.
- Register R14, R15 contents are unknown and other registers are preserved across this macro call.

C:

- Returns a pointer to the area of 64-bit ECB heap storage that was reserved.

Exception:

CTL- 7B	Heap storage exhausted.

Programming Consideration:

- Any valid register for an ECB controlled program (R0-R7, R14 or R15) can be used for size parameter.

- The amount of heap-resident storage available for use as ECB-unique storage is determined during installation. Check with the responsible party if the maximum is in doubt. This limit applies to the sum total of allocation requests issued by the ECB, not just to any single request.
- In addition to the normal macro trace information the macro trace entry will contain the size, in number of bytes, and address of the allocated storage.

Example:

Assembler:

```
LA R4,100

LA R5,2

CALOC COUNT=R5,ESIZE=R4
```

Allocates 200 bytes of heap storage, initializes to zeros and returns the address in register R5.

C:

```
soapFault *buff;
buff=calloc(1,sizeof(soapFault));
```

Allocates heap storage dynamically based on the buff elements.

FREEC – Releases storage blocks.

Release storage blocks obtained using MALOC/CALOC back to heap.

Syntax:

Assembler:

FREEC BLOCK=Rn

BLOCK	***Rn*** – General purpose register (R0-R7, R14 or R15) containing the address of the storage block to be released.

C:

void free(void*);

Other variants in C: None

Entry Conditions:

- The specified register must contain a valid storage block address of heap storage allocated earlier.

Return Conditions:

Assembler:

- Register R14, R15 contents are unknown and other registers are preserved across this macro call, expect for the register specified in BLOCK parameter.

C:

- void.

Exception:

CTL-74F	Invalid heap address specified.

Programming Consideration: None

Example:

Assembler:

```
FREEC BLOCK=R5
```

Releases the previously allocated heap Storage pointed by Register R5.

C:

```
free(ptr);
```

Releases the previously allocated heap pointed at pointer ptr.

EHEAPC – Manage ECB heap storage

- Create tagging information to be associated with a particular entry control block (ECB) heap buffer
- Locate an ECB heap buffer address
- Get 31-bit ECB heap allocation information
- Obtain the current heap check mode setting for the process
- Disable heap check mode for the process.

Syntax:

Assembler:

EHEAPC FUNC=CREATE/INFO/LOCATE,HPCKQUERY,HPCKDISABLE,TAG=reg,

ADDR=reg1,RC=reg2

FUNC	specifies the requested ECB heap action, where: **`CREATE`** creates tagging information to be associated with the specified ECB heap buffer address. **`INFO`** returns information about 31-bit ECB heap space usage. Use this option to debug dynamic memory allocation and track ECB heap space usage. **Note:** 1. You cannot use this value to obtain information about the 64-bit ECB heap. 2. If additional 31-bit ECB heap buffers are allocated or released after calling the EHEAPC macro with FUNC=INFO specified, subsequent calls to this macro may produce different results. **`LOCATE`** locates an ECB heap buffer address from the specified unique tag.
HPCKQUERY	specifies that the current heap check mode setting for the process is to be returned.
HPCKDISABLE	specifies that heap check mode should be disabled for the process.
TAG	specifies a tag identifier based on the action specified with the FUNC parameter, as follows: • If you specify FUNC=CREATE, this parameter specifies a 32-byte identifier to be associated with the specified ECB heap buffer address. The first 8 bytes of the identifier must be unique for this process. • If you specify FUNC=LOCATE, this parameter specifies a unique 8-byte identifier that is used to locate the desired ECB heap buffer address. Specify one of the following: ***`reg`*** is a general register that contains the address of the identifier. ***`label1`***

	is the label of an area that contains the identifier. The tag identifier is intended to provide additional diagnostic information for an ECB heap buffer; for example, a description of the buffer usage. The identifier is assumed to contain printable EBCDIC characters. This tag will be included in system error dumps when ECB heap buffers are included in the dump output. If the tag does not contain printable EBCDIC characters, it will not be readable in the formatted dump output. **Note:** Tag identifiers that begin with the letters I, i, TPF, TPF_, tpf, and tpf_ are reserved for future use by IBM®.
ADDR	specifies one of the following: • If you specify FUNC=CREATE, specify the general register that contains the ECB heap buffer address to which the specified tag identifier will be associated. • If you specify FUNC=INFO, specify the general register that contains the starting address of a parameter block as defined by the IEHINF data macro. • If you specify FUNC=LOCATE, specify the general register that will contain the ECB heap buffer address on return.
RC	specifies the general register that will contain the results of the requested action on return.

C: None

Entry Conditions:

- Register 9 (R9) must contain the address of the ECB that is being processed.

Return Conditions:

Assembler:

One of the following:

- If you specify FUNC=CREATE, the register specified with the RC parameter contains one of the following:

  `EHEAPC_OK`

  The request was completed successfully.

  `EHEAPC_DUPETAG`

  The TAG parameter contains an identifier that already exists for this process.

  `EHEAPC_ADDR_NOT_VALID`

  The ADDR parameter contains an address that does not represent a valid ECB heap buffer address.

  `EHEAPC_TAG_EXISTS`

  The ECB heap buffer specified on the ADDR parameter already has a tag identifier associated with it.

  `EHEAPC_TAG_NOT_ADDRESSABLE`

  The TAG parameter does not indicate a valid EVM address.

- If you specify FUNC=LOCATE, the register specified with the ADDR parameter contains the address of the ECB heap buffer associated with the specified tag identifier. If the specified tag identifier cannot be found, the register contains 0.
- If you specify FUNC=INFO, the parameter block indicated by the register specified with the ADDR parameter is initialized with information about the current usage of the ECB heap. The following fields are defined in the IEHINF data macro:

  IEHINF_ARENA

  The total size, in bytes, of the virtual address space allocated for the 31-bit ECB heap. This includes space allocated for the 31-bitpreallocated ECB heap.

  IEHINF_ORDBLKS

  The maximum number of 1-MB frames that this ECB is allowed to obtain to dynamically satisfy 31-bit ECB heap requests.

  IEHINF_SMBLKS

  The total number of 4-KB units that are assigned to this ECB for the 31-bit preallocated ECB heap.

  IEHINF_HBLKS

  The total number of 4-KB units that are assigned to this ECB for the ECB heap control table.

  IEHINF_HBLKHD

  The total size, in bytes, of the main storage allocated to the ECB heap control table.

  IEHINF_USMBLKS

  The total size, in bytes, of the main storage currently in use in the 31-bit preallocated ECB heap.

  IEHINF_FSMBLKS

  The total size, in bytes, of the main storage currently available for use in the 31-bit preallocated ECB heap.

  IEHINF_UORDBLKS

  The total size, in bytes, of the main storage currently in use in the 31-bit ECB heap, excluding the 31-bit preallocated ECB heap.

  IEHINF_FORDBLKS

  The total size, in bytes, of the main storage currently available for use in the 31-bit ECB heap, excluding the 31-bit preallocated ECB heap.

  IEHINF_KEEPCOST

  The total number of 1-MB frames currently attached to this ECB and being used for the 31-bit ECB heap.

  In addition, the register specified with the RC parameter contains one of the following:

  EHEAPC_OK

  The request was completed successfully.

  EHEAPC_ADDR_NOT_VALID

  The ADDR parameter contains an address that does not represent a valid storage address.
- When this macro is called from the control program (CP), the contents of R7 are not saved.
- If you specify FUNC=HPCKQUERY, the register specified with the RC parameter contains one of the following:

  EHEAPC_HEAPCHECK_ON

  indicates that the process is currently running with heap check mode enabled.

EHEAPC_HEAPCHECK_OFF

indicates that the process is currently running with heap check mode disabled.

Exception: None

Programming Consideration:

- The EHEAPC macro does not actually reserve an ECB heap buffer. To allocate an ECB heap buffer, use the CALOC, MALOC, or RALOC macro.
- When you code this macro in the CP, you can specify only R0–R6 for the parameters that specify a register.
- If an ECB heap buffer is resized using the `realloc` or `realloc64` function, or the RALOC macro, the associated tagging information is carried to the new ECB heap buffer address.
- The EHEAPC macro with the FUNC=HPCKDISABLE parameter is intended for use only by those applications that use large amounts of ECB heap storage and that routinely run out of available memory in the ECB heap when heap check mode is enabled.
- The EHEAPC macro with the FUNC=HPCKDISABLE parameter can be used to remove the process from heap check mode at any point during the life of the process, and thus potentially reduce the amount of ECB heap storage required by the process. However, for best results, it is recommended that this be done as early in the process life cycle as possible to reduce the amount of fragmentation that can occur in the ECB heap if allocations are initially performed when the process is running in heap check mode.
- If the EHEAPC macro with the FUNC=HPCKDISABLE parameter is specified and the process is not currently running in heap check mode, control is returned immediately.
- The EHEAPC macro with the FUNC=HPCKDISABLE parameter requires that the issuing program be authorized to issue restricted macros.

Example:

Assembler:

The following example allocates an ECB heap buffer and associates it with the specified tag.

```
   LA     R1,100                      Set buffer size to 100
   MALOC  SIZE=R1
   EHEAPC FUNC=CREATE,TAG=TAGNAME,ADDR=R1,RC=R15
   CGHI   R15,EHEAPC_OK               Did we tag the buffer ok?
   BNE    EHEAPC_NOK                  No, go process the error
EHEAPC_OK DS   0H
....
....
EHEAPC_NOK DS   0H
....
....
TAGNAME     DC     CL32'ITEST001.SORT.TABLE.BUFFER'
```

DECBC – Manage data event control blocks

Use this general `macro to perform the following actions:

- Create a new data event control block (DECB)
- Locate an existing DECB
- Swap a core block between a DECB and an entry control block (ECB) data level
- Release an existing DECB that is no longer in use
- Determine if a given address references a valid DECB that is in use.

Syntax:

Assembler:

DECBC FUNC=CREATE/LOCATE/RELEASE/SWAPBLK/VALIDATE,DECB=REG,

NAME=REG,CHAIN=INUSE/ANY,WKREG=REG,AMODE=AMODEDEF/31/64,

LEVEL=LEVEL,ERROR=LABEL3,RC=REG

FUNC	specifies the requested DECB action. The following values are allowed: **`CREATE`** creates a new DECB. **`LOCATE`** locates a previously allocated DECB with a name matching the value specified in the NAME parameter or locates the next DECB in the chain specified by the CHAIN parameter. **`RELEASE`** releases the DECB referenced in the DECB or NAME parameters (if the DECB is not currently in use). **`SWAPBLK`** Exchanges core block and related information between a DECB and a specified ECB data level. The file address information is not swapped. **`VALIDATE`** determines if the DECB specified is a valid DECB.
DECB	is used for FUNC=CREATE and FUNC=LOCATE, specifying the label or general register where the address of the DECB is to be returned. For all other actions, specifies the label or general register containing the address of a valid DECB for which the specified action is to be performed. For E-type programs, allowable register values are R1–R7. For control programs (CPs), allowable register values are R1–R7, R14, or R15.
NAME	specifies the general register containing the address (or a label indicating the address) of a 16-byte name associated with a DECB. For E-type programs, allowable register values are R0–R7. For CPs, allowable register values are R0–R7, R14, or R15.
CHAIN	specifies the next DECB entry to be returned. **`INUSE`** returns the next active DECB chained to the current DECB. The

	register specified by the DECB parameter indicates the address of the currentDECB. If the register specified by the DECB parameter contains a value of zero, it returns the first active DECB. **ANY** returns the next allocated DECB without first verifying that the DECB is active. The register specified by the DECB parameter indicates the address of the current DECB. If the register specified by the DECB parameter contains a value of zero, the first DECB will be returned without verifying the active state of the DECB.
WKREG	specifies an available general register (R1–R7, R14, or R15) to be used on the macro call. If no register is specified, the default register is R14.
AMODE	specifies the format that is used by all address parameters. The valid address format values are: ***amodedef*** The default value is specified by the AMODEDEF parameter of the BEGIN macro. If AMODEDEF is not specified on the BEGIN macro, the default value is 31. **31** indicates that addresses are specified in 31-bit format. **64** indicates that addresses are specified in 64-bit format.
LEVEL	specifies a valid ECB data level (D0–DF).
ERROR	is the label in which to branch if an error occurs during processing of the macro. If ERROR is not specified and an error occurs, control will return to the next sequential instruction (NSI). The caller then checks the contents of the register specified on the RC parameter for the associated return code.
RC	specifies the general register (R0–R7, R14, or R15) where the return code is placed when the macro ends. For E-type programs, R15 must be specified.

C:

```
TPF_DECB  *tpf_decb_create(char *name, DECBC_RC *rc);
```

NAME	A pointer to a 16-byte user-specified DECB name. The name parameter is optional. If NULL is coded, a name is not assigned to the DECB.
RC	A pointer to the return code. **rc** is an optional parameter and, if NULL is coded, the return code will not be set.

Other variants in C:

```
TPF_DECB  *tpf_decb_locate(char *name, DECBC_RC *rc);
```

```
void tpf_decb_release(TPF_DECB *decb);

void tpf_decb_swapblk(TPF_DECB *decb, enum t_lvl level);

DECBC_RC tpf_decb_validate(TPF_DECB *decb);
```

Entry Conditions:

- R9 must contain the address of the ECB being processed.

Return Conditions:

Assembler:

- When called from an E-type program, the contents of R14 and R15 cannot be predicted. The contents of all other registers are maintained.
- The register specified on the RC parameter contains the return code from one of the following:

  DECBC_OK

  The request completed successfully.

  DECBC_DUPNAME

  The NAME parameter specified for FUNC=CREATE contains a DECB name that already exists for this ECB. The address of the existing DECB is returned in the register or storage location specified by the DECB parameter.

  DECBC_NOTFOUND

  A DECB matching the input search criteria could not be found when specifying FUNC=LOCATE.

  DECBC_NOTVALID

  The DECB specified for FUNC=VALIDATE did not reference a valid DECB address or the DECB is no longer in use.

C:

- A pointer to a DECB and the return code is set to DECBC_OK.
- A NULL pointer and **rc** contains DECBC_DUPNAME if the name parameter specified contains a DECB name that already exists for this entry control block (ECB).

Exception: None

Programming Consideration:

- The DECB is an alternative to standard ECB data level information. Data level information is used to specify information about I/O request (core block reference word (CBRW) and file address reference word (FARW)) fields. The DECB fields specify the same CBRW and FARW information without requiring the use of a data level in the ECB. All the same requirements and conditions that apply to the CBRW and FARW in the ECB also pertain to the same field information in the DECB.
- The FARW in the DECB provides storage for an 8-byte file address. The DECB layout is in the IDECB DSECT (see the IDECB DSECT for more information about the fields in

a DECB).

- FUNC=SWAPBLK is not intended to perform the same operation as the FLIPC macro, which is for ECB data levels. When FUNC=SWAPBLK is specified, only the CBRW information is exchanged between the specified DECB and ECB data level.
- Macros that support the use of a DECB (such as FILEC, FINDC, and others) will only accept a DECB address as a valid reference to theDECB. If an application does not maintain the address of a particular DECB and instead maintains the name of the DECB, the caller will first have to issue the DECBC macro with the FUNC=LOCATE parameter to obtain the address of the desired DECB. The resulting address of theDECB can then be passed to the subsequent macro call.
- FUNC=LOCATE specified with the CHAIN parameter is intended for use by system utilities that have a requirement to step through each DECBassociated with a particular ECB. This parameter must only be used when the application is running in the ECB virtual memory (EVM).
- The letters I, i, TPF, TPF_, tpf, and tpf_ are reserved for future use by IBM® for DECB name spaces.

Example:

Assembler:

The following example obtains a DECB and assigns it a name of TESTDECB. The address of the newly created DECB is returned in register 7 (R7).

```
DECBC     FUNC=CREATE,DECB=(R7),NAME=NEWDECB
NEWDECB  DC     CL16'TESTDECB'
```

C:

The following example creates a DECB with a given name.

```
DECBC_RC    rc;
TPF_DECB    *decb;
char        decb_name[16] = "APPLWXY";
⋮
if ( (decb = tpf_decb_create(decb_name, &rc)) != NULL );
{
                        /* DECB is successfully created */
} else
{
      /* failed to create DECB, check rc for the reason */
}
```

Q1. In which case Block check mode will fail?

Q2. Explain different ways to get 120 bytes of work area?

Q3. Explain different ways to release block?

Q4. Explain different ways to save core area before use?

Q5. What is an advantage of CALOC over GETCC?

Q6. What are the differences between CALOC and MALOC?

Q7. What are the differences between CRUSA and RELCC?

5. File Management

5.1 Introduction to File Management

- zTPF supports two types of secondary storage devices:
    - Direct Access Storage Device (DASD)
    - (Magnetic) Tapes
- All the online databases in zTPF are maintained in the DASD.
- The online database of the zTPF system is allocated at system generation time and is shared by all entries and all applications
    - Fixed file storage.
    - Pool files storage.
- Magnetic tapes are used for online and offline processing and can be used for both input and output in the z/TPF system.
- General files and general data sets are usually related to some offline processing. Data is either produced online to be processed offline by z/OS programs or data is produced offline for online processing. Both structures provide a data interface between the offline and online system components.

5.2 zTPF Online Database

- The online database is the main database in the z/TPF system and its organization of data is unique to the z/TPF system
- Data is stored and retrieved at the physical record level. The records can be classified according to the following sizes:
    - Small (381 bytes)
    - Large (1055 bytes)
    - 4K (4095 bytes).
- z/TPF files are allocated across the range of physical storage media to balance and, therefore, improve access performance. Therefore, successively numbered records are allocated to different physical direct access storage devices (DASD).
- The structure of the online data is identified during system generation. Data is dynamically generated; however, the facilities for storing the data are identified prior to the online operation.
- When a DASD is installed, it is formatted to define the number of each category of record to be stored.
- Each record in the system is physically identified in the DASD using a Module number, Cylinder number, Head number and a Record number (MCHR).
- This information is encoded in symbolic form referred to as ‘File Address’.

5.2.1 Fixed File

- **Fixed files** are records defined before being referenced by application program.
- These are mostly served as indices to some other dynamic data.
- A set of fixed file records identified to belong to a record type.
- Each fixed file in a record type identified with a record type ordinal number.
- A fixed file record is uniquely identified by its Record type and ordinal number.
- New fixed file cannot be created during application execution.
- Fixed records are allocated in the online database so as to ensure faster access and improved performance.
- Application programmer defines the record type, number ordinals and organization of the record but database administrator will allocate the fixed file by application programmer request.
- Record types can be either specific or Miscellaneous.
- Miscellaneous record types used for application request that don’t require more than a few ordinals, the Miscellaneous record types that are predefined are:
- #MISCS, #MISCL, #MISC4, for 381 byte, 1K and 4K records respectively.
- System equates defined to associate a set of fixed file records to a specific record type in the Face table (FCTB).
- The Sys equate along with a ordinal number is used by application programs to get to a particular fixed file.

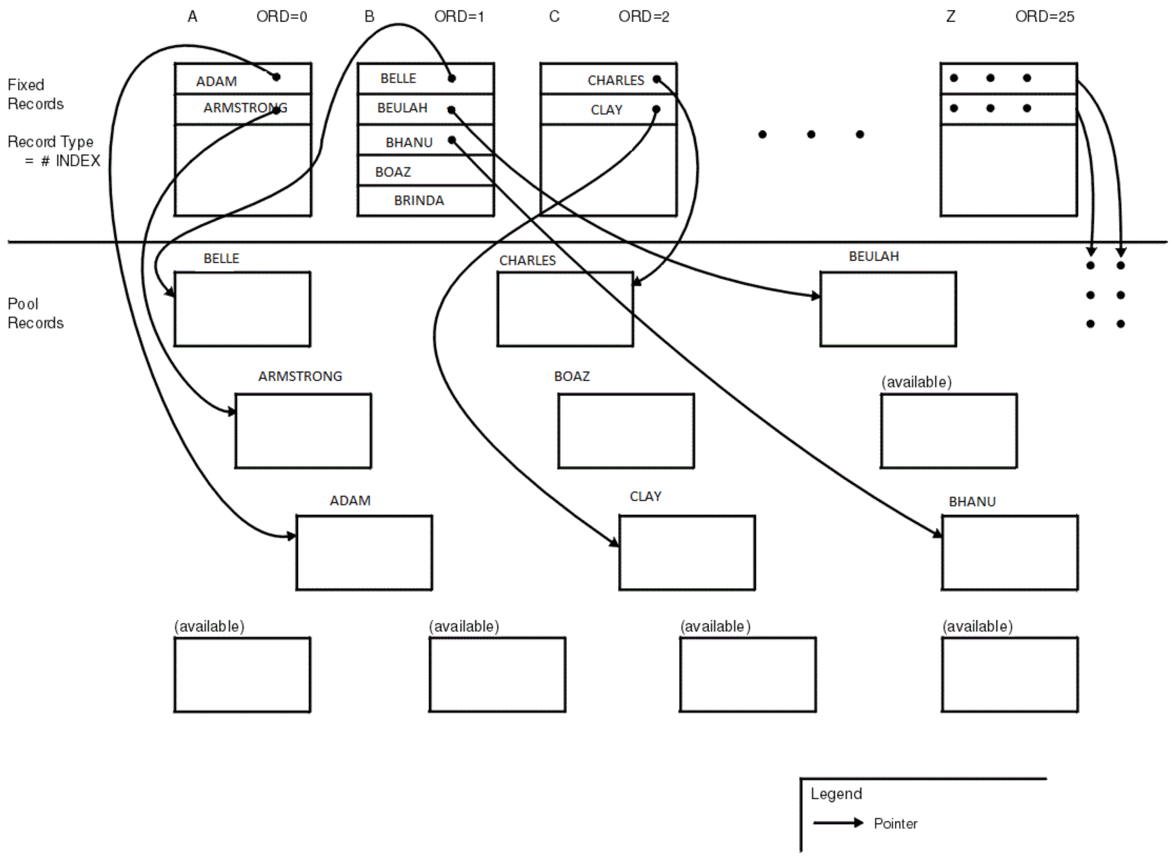
5.2.2 Pool File

- The dynamic requirements for file storage are satisfied by the use of areas on modules known as **pools.**
- **The pool file area is shared by all applications.**
- Several pool record types are defined by the system, based upon record attributes in order to allocate pool areas on modules.
- The records in pool file storage, or just pool, are dispensed as needed and returned to the system when they are no longer needed and can be re-dispensed.
- When a pool record is requested by an application program, the system locates an available record and returns a pointer (address) to the pool record; typically the pointer is saved in a fixed file record for subsequent reference.
- They are dispensed on a need basis. When requested, the system allocates an available file from a pool and returns the symbolic file address to the application.
- The pool files are classified into two categories based on its life time:
    - Long term pool file
    - Short term pool file

- The long term pool files are used to store data of permanent nature.
- They are released back to the system only when the application requests for it.
- Typically, the address of a long term file is stored in a fixed file for subsequent reference.
- The Short term pool file is used to store data of temporary nature.
- The system automatically recycles them without any consideration, if does not have any to dispense. Therefore it has to be used with caution.

5.2.3 Illustration of a Card member Database-Fixed and Pool Files

- For instance, to develop a database that maintains the relation between **card member name** and the **details of each customer**, a fixed file with 26 ordinals can be used. **Record type** (in this case, #INDEX) and **ordinal numbers** (0 through 25).
- In order to retrieve (that is, to find or read) a pool record, the application must know its address. The most common technique used by applications is to form data structures that use fixed records as indexes to pool records.
- The index, a **fixed file record** is retrieved by using the first letter of the last name of the cardmember name to calculate the address of the appropriate index record. The index is then scanned for the name, and, when that item is found, the associated customer detail file embedded in that item, which is a **pool address** is accessed. This allows the selection of one of a vast number of pool records with a minimum of file accesses (that is, I/O operations)
- Below Diagram illustrates the above mentioned point.

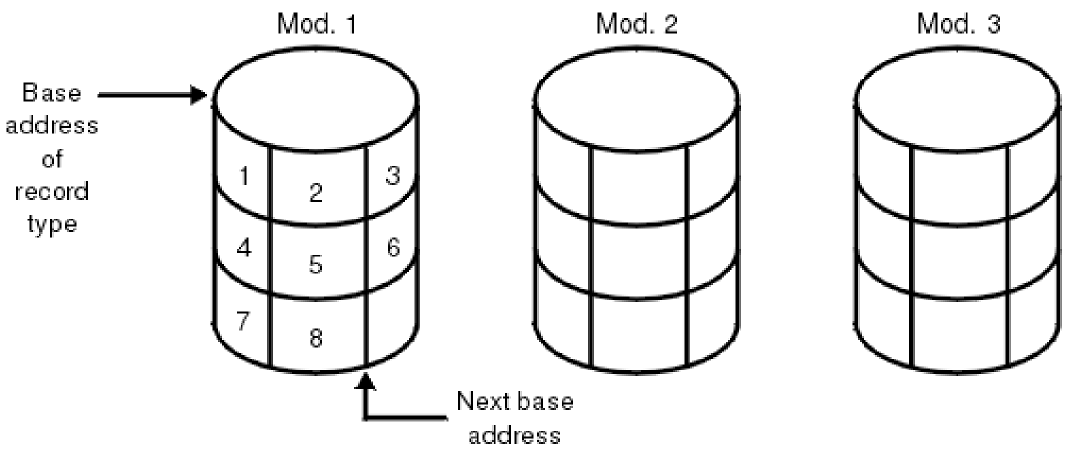


- Let us say an ordinal("S") does not have enough space to accommodate a new name, a pool file is requested and the new information is stored in the pool file.

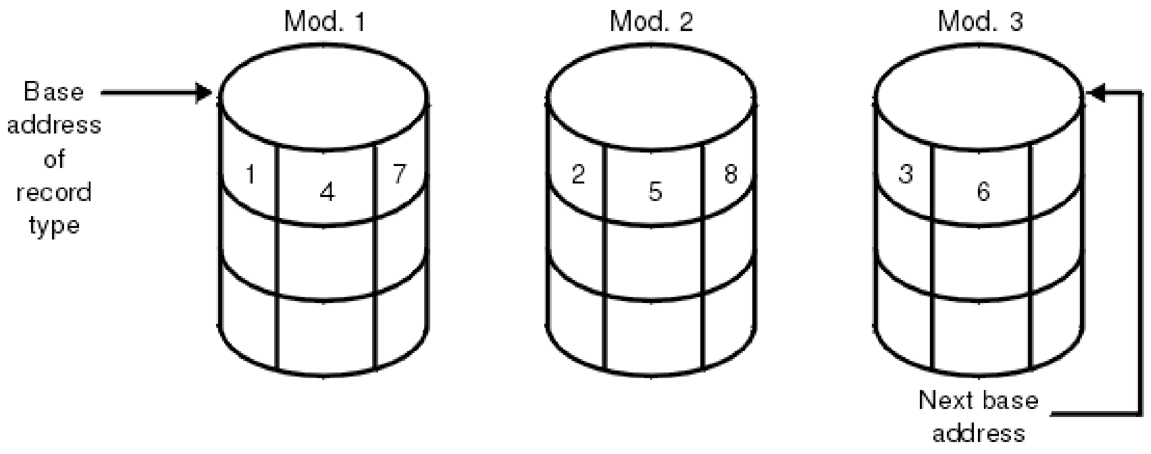
- The pool file is then chained to the ordinal (a fixed file). The fixed file is called the **prime** and the chained pool file(s) are called as **overflows**.

5.2.4 Database Allocation

- **Vertical Database Allocation:** Records in general file or general data set are contained on one physical storage medium and are allocated in a sequential fashion similar to a conventional z/OS sequential data set.



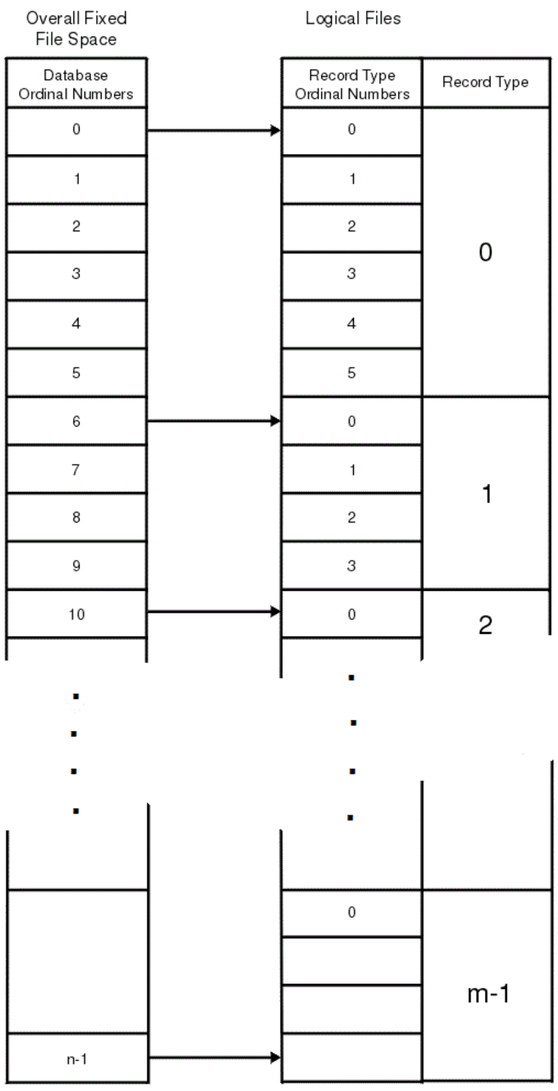
- **Horizonatal Database Allocation:** The organization of data in the online z/TPF system is unique to the z/TPF system. z/TPF files are allocated across the range of physical storage media to balance and, therefore, improve access performance. Therefore, successively numbered records are allocated to different physical direct access storage devices (DASD).



5.2.5 DBON and physical address

- The z/TPF system views the entire file space as a repository for holding 4K, 1055-byte and 381-byte records
- Within a given record type (for example, 4K fixed records), the z/TPF system has several database ordinal numbers (DBON)
- Consider the representation of the fixed file space of n records (database ordinal numbers 0 through n-1). The number of record types is given as m, where m is less than n.
- To allocate fixed file space means dividing the entire fixed file space into subgroups of record types, where a subgroup consists of either all small, all large, or all 4K fixed records. The allocation is accomplished by creating a FACE table that associates the base address, called a **beginning database ordinal number**

(DBON), with each record type. For example, in the below diagram, record type 1 begins at DBON 6 and four records are allocated to record type 1. The next record type (record type 2) begins at DBON 10. In general, each record type is associated with some **unique DBON**.

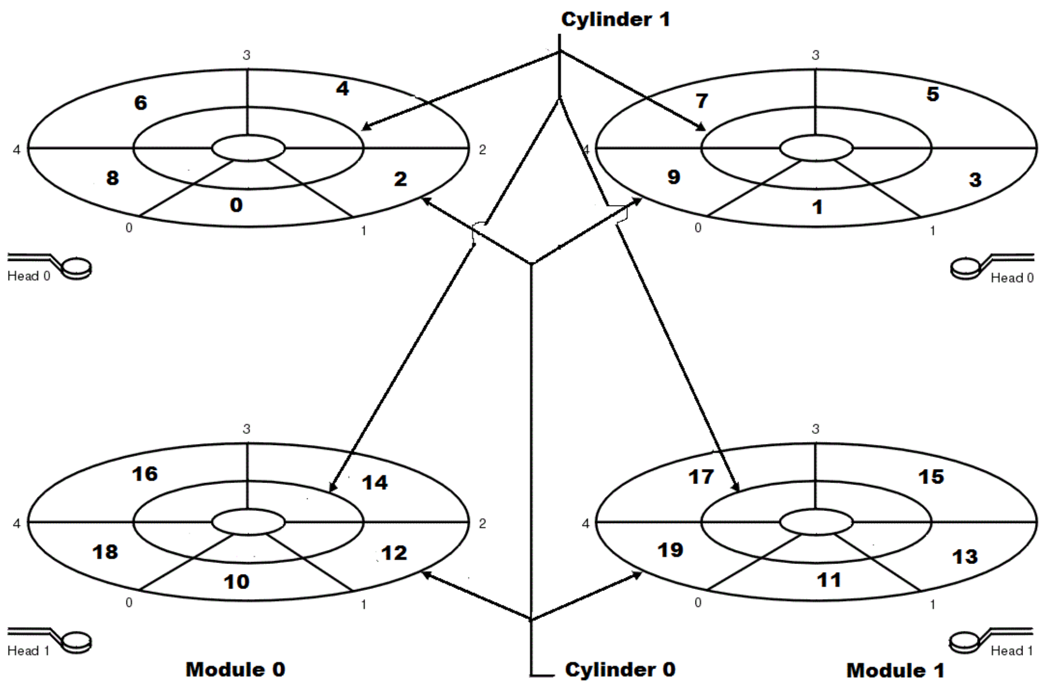


- A fixed file area for holding record types consisting of records of the same size has a set of database ordinal numbers (0 through n-1).
- Each fixed record type has a set of DBONs equal to the number of records designated for that record type.
- There is a similar procedure for allocating the pool file space. The record type for a pool file is based on the pool record type (such as SSTx and LDPx).
- The online physical file storage is allocated for the different record sizes in proportions that are primarily determined by application design. The sum of allocations of all pool and fixed records comprises the entire online physical file space.
- The z/TPF system ensures that the sequential database ordinal numbers (DBON) do not map into physically adjacent records. It is assumed, in general, that the

horizontal allocation of records of a given record type throughout the file storage permits a greater chance for simultaneous accessing than the more conventional, vertical allocation,

- The allocation of records associates successive database ordinal numbers (DBON) with successive **modules, cylinders, heads, and records** on modules.

Consider the below example that shows a configuration of two modules, two cylinders, two heads per module, and five records per track. The numbers in bold and larger font size represent **database ordinal numbers (DBON).** The numbers in smaller font size represent the **device addresses,** that is, the names of physical locations.



5.2.6 Record Duplication

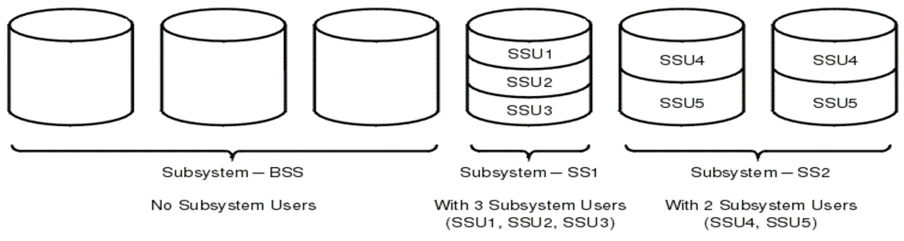
- In the z/TPF system, data records (on modules) can be duplicated, which means that there are two copies of a data record on the database. The copies are referred to as the **primary record** and the **backup record**, sometimes called the **duplicate record or dupe**
- Record duplication addresses the design objectives of the z/TPF system of:
    - Performance
    - Database integrity

  by providing the ability to maintain two copies of critical data records.
- Primary and backup records must be allocated to the same device type. Allocation of fixed and pool data can be specified as:
    - **Non-duplicated** - None of the records are duplicated on the device type

    - **Selectively duplicated** - Selected (critical) records are duplicated on the device type
    - **Fully duplicated -** All records (fixed and pool) are duplicated on all of the device types generated in the user installation.
- The allocation procedure assigns two database ordinal numbers (DBONs) to a duplicated record, each to different modules.
- The duplicate record in a record type (either pool or fixed) is assigned to the same relative position on an alternate disk module. Therefore, an even number of modules is required.

5.2.7 Multiple Database function (MDBF) overview

- The multiple database function (MDBF) provides the user with the ability to:
    - Physically separate sets of data
    - Logically separate sets of data
    - Physically and logically separate sets of data.
- The use of MDBF introduces the concepts of subsystems and subsystem users.
- In the following diagram the Subsystem provides a physical separation of data, while a Subsystem user provides a logical separation of data.



5.2.8 Data record Attributes

- **Physical residence:** the physical location of a record is transparent to the application environment. The z/TPF system manages the following physical forms of storage on which data records can reside – Modules, Magnetic tape, Main Storage.
- **Device Type:** The z/TPF system supports four device types (DEVA, DEVB, DEVC, and DEVD). Each of these device types represents distinct characteristics for accessing and managing the location of data on the online DASD subsystem. Data is mapped across the DASD subsystem based on the several factors, including:
    - Relative starting module number
    - Number of modules defined for the device
    - Number of records per track

  - Number of tracks per cylinder
  - Number of cylinders per module
  - Module duplication factor (simples or duplex)
  - Physical record sizes; that is, small (381 bytes), large (1055 bytes), 4 KB (4096 bytes).
- **Record Size:** The logical size of a record within the z/TPF system is always one of three sizes: small (381 bytes), large (1055 bytes), 4 KB (4096 bytes).
- **Record Duplication:** In the z/TPF system, data records (on modules) can be duplicated, which means that there are two copies of a data record on the database. The copies are referred to as the primary record and the backup record, sometimes called the duplicate record or dupe
- **Record Longevity:** By definition, fixed records are maintained by the z/TPF system forever while pool records are maintained for an interval of time, which is determined by the application environment (in terms of seconds, hours, days, weeks, or months). There are short term pool records and long term pool records.
- **Pool record types:** Pool records are classified according to the attributes of size, duplication, longevity, and residence (type of device). The basic pool types are:

  - Small short-term pool (SSTx)
  - Small long-term (not duplicated) pool (SLTx)
  - Small long-term duplicated pool (SDPx)
  - Large short-term pool (LSTx)
  - Large long-term (not duplicated) pool (LLTx)
  - Large long-term duplicated pool (LDPx)
  - 4K short-term pool (4STx)
  - 4K long-term (not duplicated) pool (4LTx)
  - 4K long-term duplicated pool (4DPx)
  - 4K long-term duplicated FARF6 pool (4D6)
- **Types of Fixed file records**: Fixed file records can be classified according to the following sizes:

  - Small (381 bytes)
  - Large (1055 bytes)
  - 4K (4095 bytes).

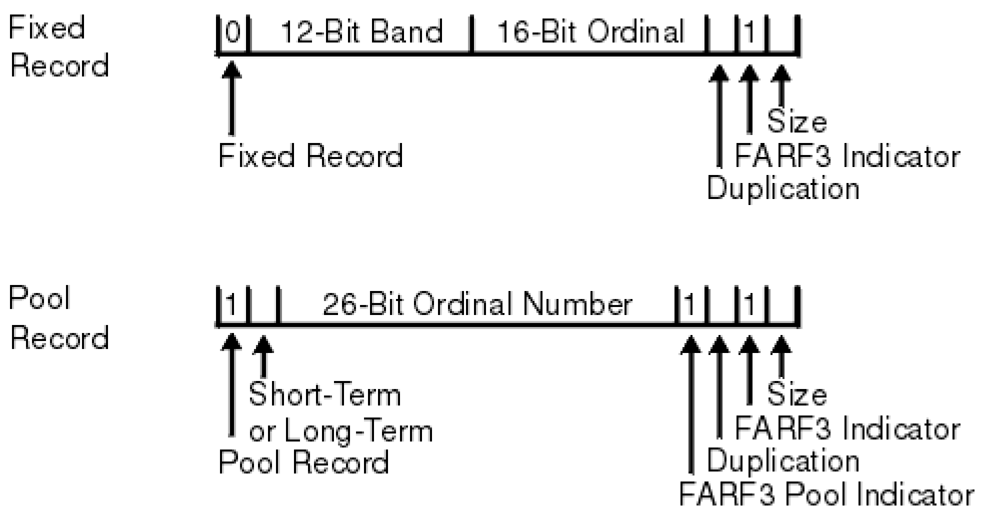
5.2.9 File Address Compute Table (FCTB)

- The file address compute (FACE) table (FCTB), a system table, is a centralized source of information about file addresses. It provides the information necessary to do the following:
    - Convert a record type and ordinal number into a file addressing scheme called file address reference format (FARF). This conversion applies to both fixed-file records and pool records.
    - Convert a FARF address into its physical record address in module cylinder head record (MCHR) format.
- For fixed addresses, the FACE table is accessed by the FACE and FACS programs and the FAC8C and FACZC macro service routines.
- For pool addresses, the FCTB is accessed by the Get File Storage macro service routines.

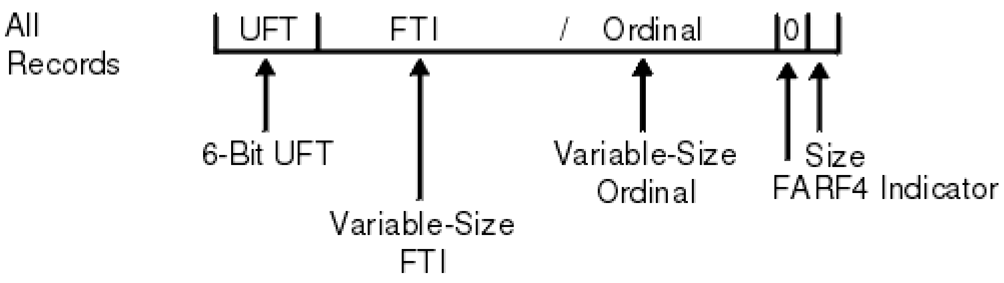
5.2.10 File Address Reference format(FARW)

- The file address reference format is the method used by the z/TPF system to symbolically address fixed and pool records.
- The 4-byte file address defined by FARF is a basic element of data organization in the z/TPF system.
- There are four file address reference formats: FARF3, FARF4, FARF5, and FARF6. These allow the z/TPF system the flexibility of adapting to module technology.

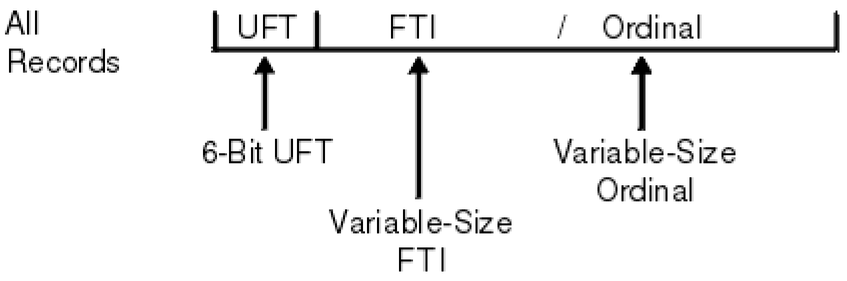
FAR3:



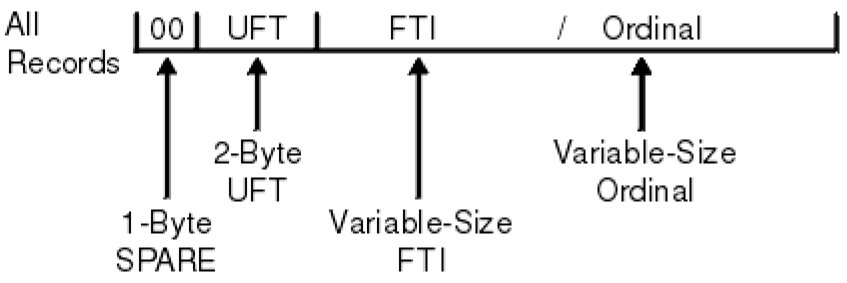
FARF4



FARF5



FARF6

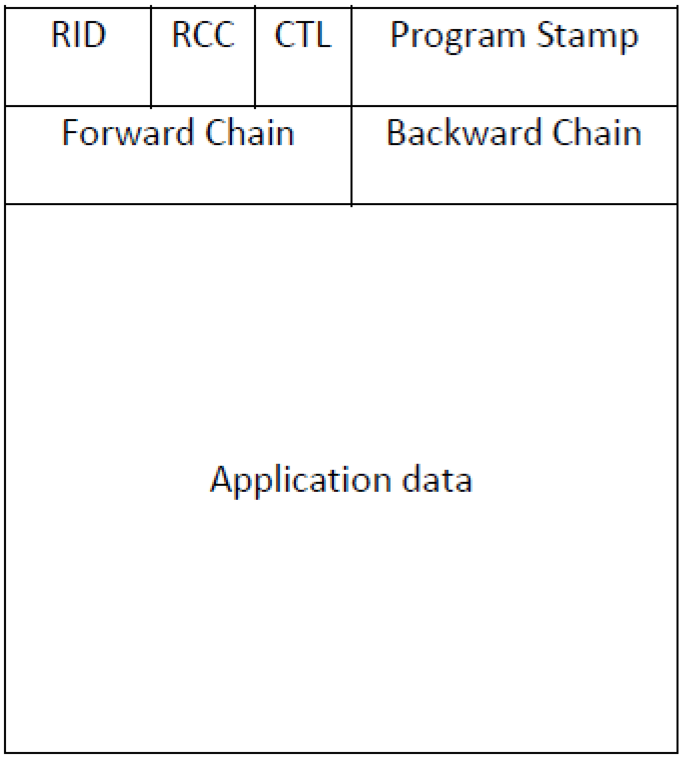


5.2.11 zTPF Standard Record Header

- Each file in the system has a standard header before the data section.
- The standard header is 8-bytes long and following are the contents:

Record ID	2 bytes
Record code check	1 byte
Control byte	1 byte
Program stamp	4 bytes

- Optionally, the forward and back ward chain (4 bytes each) are also maintained in some records.



- Every unique file in the system, fixed or pool, is associated with a 2-byte **record ID**.

- The record ID describes the characteristics of the record such as size, longevity, VFA candidacy etc.

- These characteristics are maintained in the zTPF system in a table called the **Record ID attribute table (RIAT)**.

- The **Record code check** (RCC) is a 1-byte value generated and stored by an application program. It is used to ensure and validate the integrity of the database.

- The **Control byte**, If used, will have the following set:

Bit 0 = 1	Forward chaining is applicable
Bit 1 = 1	Backward chaining is applicable
Bit 2 = 1	L2 Record
Bit 4 = 1	Overflow record

- The **Program stamp** is the last program that updated the record.

5.2.12 Record ID Attribute Table (RIAT)

- There is one item in the RIAT for each record ID defined within any subsystem, and there is a RIAT for each subsystem
- The attribute information for each ID consists of the following data:
  - Record category (0–9)
  - Exception recording, logging, and restoring data
  - User exit data
  - VFA candidacy and delay file threshold
  - Record caching candidacy
  - Lock maintenance in a loosely coupled environment.

5.3 Accessing a File

- An ECB has to fetches a file in to the main memory from the DASD in order to access it.

- To retrieve a file, the zTPF system needs to be provided with the symbolic file address of the same.

- A fixed file is uniquely identified using its **record type and ordinal number**.

5.3.1 Fixed File Access

- The steps involved are as below for a fixed file:
  - **Step 1** - Obtain the symbolic file address.
  - This is done using the FACE program which access the File address Compute Table (FCTB). The FCTB contains information to convert a record type and ordinal number to the corresponding symbolic addresses and from that to a physical address in MCHR.
  - The inputs to the FACE program are:
  - R6 should contain the record type
  - R7 should contain the FARW address of the data level where file is to be fetched.
  - R0 should contain the ordinal number.
  - FACE program when successful, returns the file address on the FARW of the specified level and the highest ordinal number for the file type on Register R0.
  - If FACE fails, then Register R0 will be zero.
  - Following is an example, to access ordinal 2 of a fixed file with record type #MISCL.

```
LA     R6, #MISCL        RECORD TYPE
LA     R0, 2             ORDINAL NUMBER
LA     R7, CE1FA0        FARW ADDRESS (D0)
ENTRC  FACE              GET FILE ADDRESS
LTR    R0, R0            ERROR?
BZ     ERROR             YES - BRANCH TO ERROR ROUTINE
```

- **Step 2** - Once the file address is obtained, the record ID and record code check have to be populated in the specified data level whose FARW contains the file address.
- **Step 3** - The final step is to issue one of the FIND type macros to retrieve the file from DASD to the core.
- The FIND macro causes the ECB to be suspended for a while until I/O completes.
- Hence it is advisable to always code a WAITC macro after issuing a FIND macro.
- On successful I/O completion, the requested File contents are copied on to a core block which is then attached to the CBRW of the specified Data level.
- If unsuccessful the following bits are set in the ECB fields CE1SUG/CE1SUD.

Bit 0 = 1	I/O Hardware Error
Bit 1 = 1	Record ID Check Failure
Bit 2 = 1	RCC Check Failure
Bit 6 = 1	Invalid File Address

5.3.2 Pool File Access

- To access a Pool file, the application needs to know the address of the file in advance.

- The file address must be populated on the FARW of the data level where file is to be fetched on.

- Then step 2 and 3 of the fixed file retrieval should be followed.

5.3.3 Record ID and RCC checks during File access

- The record ID and RCC are used to validate the integrity of a record.
- In a FIND type macro, the validation is performed by the control program as follows:

RID in FARW	Is Non-zero	The RID in FARW is compared in the header of the retrieved file. If the comparison fails, appropriate error bits are set in CE1SUD/CE1SUG fields of the ECB and the processing continues.
	Is zero	No validation is Performed

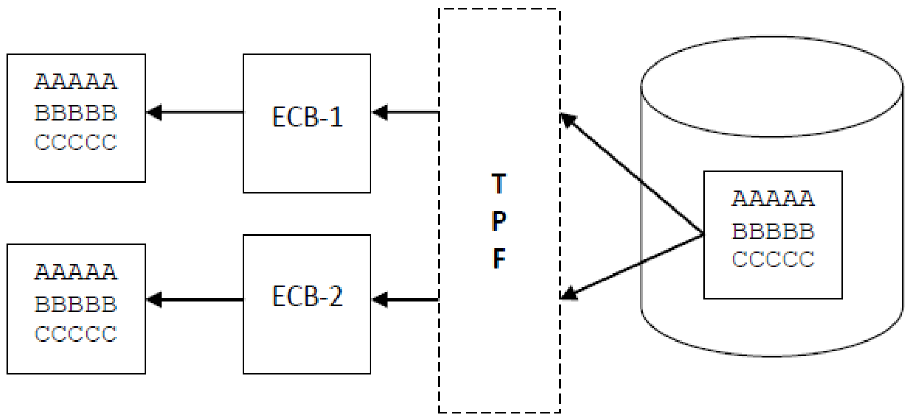
RCC in FARW	Is Non-zero	The RCC in FARW is compared in the header of the retrieved file. If the comparison fails, appropriate error bits are set in CE1SUD/CE1SUG fields of the ECB and the processing continues.
	Is zero	No validation is Performed

5.4 Updating a File

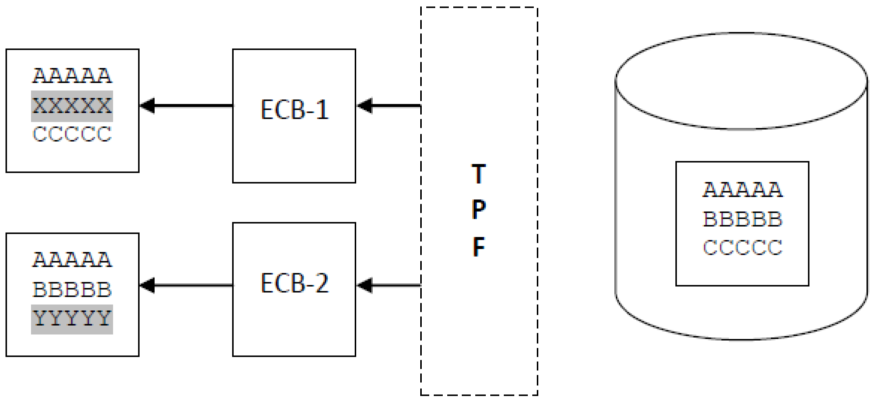
- In zTPF, a file can be updated by multiple ECBs at the same time.
- Hence, if there is no synchronization between the ECBs accessing a file, the updates made by one ECB might be over written by the other, leading the in-consistency of data.
- Following is a typical problem illustration of the phenomenon described above.

5.4.1 Problem Illustration

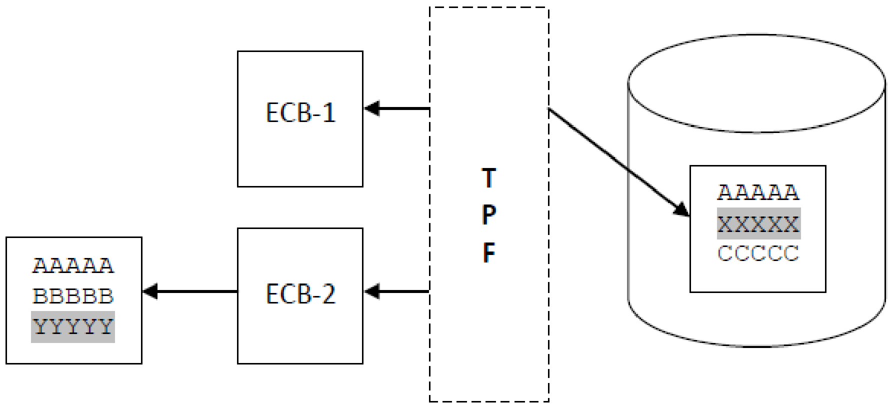
- ECB-1 and ECB-2 request for the same record simultaneously. The I/O request is serviced and both the ECBs are given a copy of the record in the core.



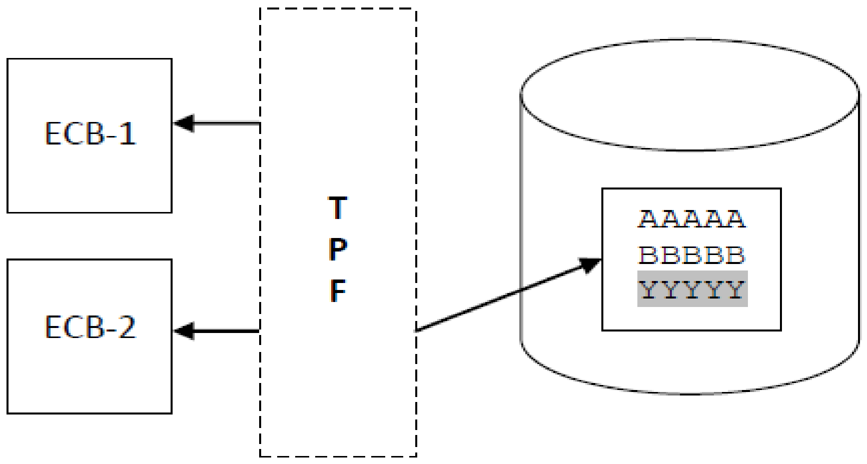
- ECB-1 and ECB-2 make their necessary updates (as highlighted) to the copies obtained by them.



- ECB-1 initiates a request to file its changes to the DASD.



- ECB -2 initiates a request to file its changes to the DASD.



- It is now evident that ECB-2 Clearly overwrites the updates of ECB-1, which leads to inconsistency of data!

5.4.2 Solution

- zTPF provides the **HOLD** mechanism to avoid such problem situations.
- The HOLD provides exclusive access to a requesting ECB. All the requests for updating the same file by other ECBs are queued.
- A record that needs to be updated must be requested with a HOLD.
- When an application is done updating the file, it must use one of the FILE macros to file the changes to the DASD (or VFA) and also release the HOLD on the record.
- This allows the next ECB waiting on queue to get access to the file.
- By principle, only the prime file is requested with HOLD.

5.4.3 Deadlock

- the HOLD mechanism might cause a deadlock situation.
- For instance, say ECB-1 which is holding a File B, tries to hold File A which is held by ECB-2 and ECB-2 is waiting to hold File B to release File A.
- To avoid such a situation, the application must be written in such a way that:

- No more than one record is on hold at any given time.

- The hold is not maintained for a time longer than needed, i.e. The hold must be withdrawn soon after the updates are complete.

5.4.4 Record ID and RCC checks during File update

- The record ID and RCC are used to validate the integrity of a record.
- In a FILE type macro, the validation is performed by the control program as follows:

RID in FARW	Is Non-zero	The RID in FARW is compared in the header of the retrieved file. If the comparison fails, a system error is generated and the ECB exits.
	Is zero	Not Allowed. RID must always be supplied for FILE macro.

RCC in FARW	Is Non-zero	The RCC in FARW is compared in the header of the retrieved file. If the comparison fails, a system error is generated and the ECB exits.
	Is zero	No validation is Performed

5.5 File management macros

FINDC – Find a File

FINDC reads a file into the core based on the parameters specified in the FARW of the provided data level.

Syntax:

Assembler:

FINDC Dn

Dn	Specifies the data level (D0-DF) on which the core block has to be assigned.

C:

void findc(enum t_lvl level);

level	One of 16 possible values representing a valid data level from the enumeration type t_lvl, expressed as Dx, where x represents the hexadecimal number of the level (0–F). The data record being retrieved is attached to this level.

Other variants in C:

void	findc_ext(enum t_lvl level,unsigned int ext);

Entry Conditions:

- The core block must not be present in the specified level.
- A file address, RID and RCC must be contained in the FARW for the specified data level.

Return Conditions:

Assembler:

- The specified data level will hold a core block.
- Register R14, R15 contents are unknown and other registers are preserved across this macro call.
- On error, CE1SUG and CE1SUD have error codes.

C:

Void

Exception:

CTL- 22	FIND issued on data level already holding a block.
CTL- 2D	Zero File Address

Programming Consideration:

- WAITC should be coded to ensure ECB processing waits until I/O operation is complete.

Example:

Assembler:

```
FINDC D2                  Finds file and puts Core block on level 2.

WAITC FINDERR             Wait for I/O to complete.
```

C:

```
findc(D6);                Retrieves data record on level 2.

if(waitc())
{
 serrc_op(SERRC_EXIT,0x12345,”I/O ERROR”,NULL);
}
```

WAITC – Suspending processing for I/O completion

Suspends the processing of ECB until I/O requests for the ECB have been serviced. If an error is encountered, the control is transferred to the label specified as a parameter.

Syntax:

Assembler:

WAITC labelerr

labelerr Label of an error routine in the current program segment.

C:

int waitc(void);

Other variants in C: None

Entry Conditions: None

Return Conditions:

Assembler:

- If a hardware error or unusual condition occurred, control is given to the error routine specified by *labelerr*.
- Register R14 and register R15 contents are unknown.
- All other registers are preserved over this macro call.
- Return to NSI if normal I/O completion.

C:

- Integer value of zero.
- Integer value representing the value of CE1SUG (gross level error indicator byte).

Exception: None

Programming Consideration:

- Single WAITC can be used to handle multiple I/O requests.
- If no I/O is pending, there is no loss of control by the entry.
- Running this macro resets the 500 ms program time-out if I/O operations are pending for this ECB.

Example:

Assembler:

```
      WAITC FINDERR                    Wait for I/O to complete.
```

C:

```
      if(waitc())                      Wait for I/O to complete.
      {
         serrc_op(SERRC_EXIT,0x1234,"cannot find f/a",NULL);
      }
```

FINWC – Find a File and wait.

Performs the operation of both FINDC and WAITC in a single macro call. Read a File into the core based on the parameters specified in the FARW of the provided data level and waits for I/O to complete.

Syntax:

Assembler:

FINWC Dn,labelerr

Dn	Specifies the data level (D0-DF) on which the core block has to be assigned.
labelerr	Label to branch to in case of error.

C:

void *finwc(enum t_lvl level);

Other variants in C:

void	*finwc_ext(enum t_lvl level, unsigned int ext);
void	*find_record_ext(enum t_lvl level, const unsigned int *address, const char *id, unsigned char rcc, enum t_act type, unsigned int ext);

Entry Conditions:

- The core block must not be present in the specified level.
- A file address, RID and RCC must be contained in the FARW for the specified data level.

Return Conditions:

Assembler:

- The contents of CBRW are unchanged and the FARW is unchanged.
- Register R14 and register R15 contents are unknown.
- All other registers are preserved over this macro call.
- On error CE1SUD and CE1SUG have error codes and controls is transferred to error path.
- All pending I/O for this ECB is completed,CE1IOC = 0

C:

- Pointer to the working storage block containing the retrieved record image.

Exception:

CTL- 22	FIND issued on data level already holding a block.
CTL- 2D	Zero File Address

Programming Consideration:

- Use FINWC instead of FINDC and WAITC.

Example:

Assembler:

```
FINWC D7,FINDERR
```

Finds file and puts Core block on CE1CR7

And wait for I/O to complete.

C:

```
finwc(D7);
```

Finds file and puts Core block on CE1CR7

And wait for I/O to complete.

FINHC – Find a File with hold.

Performs the operation of FINDC and assigns exclusive of the file to the requesting ECB.

The control program will queue this request if the record is being held by another ECB, if not the HOLD table is updated with ECB and File address. All succeeding requests to hold this record are queued until held record is released for update.

Syntax:

Assembler:

FINHC Dn

Dn	Specifies the data level (D0-DF) on which the file has to be retrieved with HOLD.

C:

void finhc(enum t_lvl level);

level	One of 16 possible values representing a valid data level from the enumeration type *t_lvl*, expressed as D*x*, where *x* represents the hexadecimal number of the level (0–F). The data record being retrieved is attached to this level.

Other variants in C:

void	*finhc_ext(enum t_lvl level, unsigned int ext);*

Entry Conditions:

- The core block must not be present in the specified level.
- A file address, RID and RCC must be contained in the FARW for the specified data level.
- The ECB must not be holding the record already.

Return Conditions:

Assembler:

- The contents of CBRW are unchanged and the FARW is unchanged.
- On successful retrieval, CE1HLD counter is incremented by 1.
- Register R14 and register R15 contents are unknown.
- All other registers are preserved over this macro call.

C:

Void

Exception:

CTL 21	Record hold table full (catastrophic error).
CTL 22	FIND issued on data level already holding a block.
CTL 2D	Zero File Address.
CTL 35	Attempt to hold same record more than once.
CTL FC	ECB attempts to hold more than 256 macros.

Programming Consideration:

- A WAITC must follow this macro.

Example:

Assembler:

```
FINHC D7            Finds file and puts Core block on CE1CR7

WAITC FINDERR       And wait for I/O to complete.
```

C:

```
finhc(D7);          Finds file and puts Core block on CE1CR7
if(waitc())         And wait for I/O to complete.
{
 serrc_op(SERRC_EXIT,0x12345,"I/O ERROR OCCURRED",NULL);
}
```

FIWHC – Find a File with hold and wait for I/O completion.

Performs the operation of FINHC and WAITC in a single macro call.

Syntax:

Assembler:

FIWHC Dn,labelerr

Dn	Specifies the data level (D0-DF) on which the file has to be retrieved with HOLD.
labelerr	Label of an error routine in the program segment.

C:

void **fiwhc(enum t_lvl level);*

level	One of 16 possible values representing a valid data level from the enumeration type *t_lvl*, expressed as D*x*, where *x* represents the hexadecimal number of the level (0–F). The data record being retrieved is attached to this level.

Other variants in C:

void	*fiwhc_ext(enum t_lvl level, unsigned int ext);

Entry Conditions:

- The core block must not be present in the specified level.
- A file address, RID and RCC must be contained in the FARW for the specified data level.
- The ECB must not be holding the record already.

Return Conditions:

Assembler:

- The contents of CBRW are unchanged and the FARW is unchanged.
- On successful retrieval, CE1HLD counter is incremented by 1.
- All pending I/O operations, including this request, are completed for this ECB.
- Register R14 and register R15 contents are unknown.
- All other registers are preserved over this macro call.

C:

- Pointer to the working storage block containing the retrieved record image.

Exception:

CTL 21	Record hold table full (catastrophic error).
CTL 22	FIND issued on data level already holding a block.
CTL 2D	Zero File Address.
CTL 35	Attempt to hold same record more than once.
CTL FC	ECB attempts to hold more than 256 macros.

Programming Consideration:

- Prefer FIWHC instead of FINHC and WAITC.

Example:

Assembler:

```
FIWHC D7
```
Finds file and puts Core block on CE1CR7 with HOLD and wait for I/O to complete.

C:

```
struct im0im *inm;
inm = fiwhc(D7);
```
Finds file and puts Core block on CE1CR7 with HOLD and wait for I/O to complete.

FILEC – File a Record

FILEC writes a file working copy in core to file and releases the core block.

Syntax:

Assembler:

FILEC Dn

Dn	Specifies the data level (D0-DF) on which the core of the file exists.

C:

void filec(enum t_lvl level);

level	One of 16 possible values representing a valid data level from the enumeration type t_lvl, expressed as Dx, where x represents the hexadecimal number of the level (0–F). The working storage block on this data level is the record to be filed.

Other variants in C:

void	filec_ext(enum t_lvl level,unsigned int ext);

Entry Conditions:

- The core block must be present in the specified level.
- A file address, RID and RCC must be contained in the FARW for the specified data level.

Return Conditions:

Assembler:

- The specified data level will no longer hold a core block.
- Register R14, R15 contents are unknown and other registers are preserved across this macro call.

Exception:

CTL- 23	No block in CBRW
CTL- 2D	Zero File Address.
CTL- 2E	RID check failure.
CTL- 2F	RCC check failure.
CTL- 30	Unable to file the record in DASD.
CTL- 32	File block and Storage block size mismatch.

Programming Consideration:

- Program stamp updated on File header.
- RID on FARW cannot be Non-zero. RCC can be zero.
- Make sure RID is same in core block and FARW.
- Do not use core block in the CBRW after FILEC

Example:

Assembler:

`FILEC D2`	*Files the record in data level 2.*

C:

`filec(D6);`	*Files the record in data level 6.*

FILNC – File a File Record with No Release

FILNC writes a file working copy in core to file but does not releases the core block.

Syntax:

Assembler:

FILNC Dn

Dn	Specifies the data level (D0-DF) on which the core of the file exists.

C:

void filnc(enum t_lvl level);

level	One of 16 possible values representing a valid data level from the enumeration type t_lvl, expressed as Dx, where x represents the hexadecimal number of the level (0–F). The working storage block on this data level is the record to be filed.

Other variants in C:

void	filnc_ext(enum t_lvl level,unsigned int ext);

Entry Conditions:

- The core block must be present in the specified level.
- A file address, RID and RCC must be contained in the FARW for the specified data level.

Return Conditions:

Assembler:

- CBRW at specified level is unavailable but is made available on return from the next WAITC call.
- Register R14, R15 contents are unknown and other registers are preserved across this macro call.

Exception:

CTL- 23	No block in CBRW.
CTL- 2D	Zero File Address.
CTL- 2E	RID check failure.
CTL- 2F	RCC check failure.

CTL- 30	Unable to file the record in DASD.
CTL- 32	File block and Storage block size mismatch.
CTL- 3F	Error occurred during FILNC processing.

Programming Consideration:

- To ensure the file was completed, WAITC needs to follow FILNC.
- Program stamp updated on File header.
- RID on FARW cannot be Non-zero. RCC can be zero.
- Make sure RID is same in core block and FARW.
- Do not use core block in the CBRW after FILEC

Example:

Assembler:

```
FILNC D2
```
Files the record in the data level 2and the file remains attached.

```
WAITC FILERR
```
Waits till the record is filed.

C:

```
filnc(D6);
if(waitc())
{
 serrc_op(SERRC_EXIT,0x12345,"I/O ERROR",NULL);
}
```
Files the record in data level 6 and remains attached.

UNFRC – Unhold a File record.

Releases the hold on the record address contained in the FARW of the specified data level.

Syntax:

Assembler:

UNFRC Dn

Dn	Specifies the data level (D0-DF) on which the held file exists.

C:

void unfrc(enum t_lvl level);

level	One of 16 possible values representing a valid data level from the enumeration type *t_lvl*, expressed as D*x*, where *x* represents the hexadecimal number of the level (0–F). This argument identifies the FARW containing the file address of the record to be removed from the record hold table.

Other variants in C:

void	*unfrc_ext(enum t_lvl level, unsigned int ext);*

Entry Conditions:

- The file must be present with hold in the specified level.

Return Conditions:

Assembler:

- CE1HLD is decremented by 1.
- FARW is unchanged.
- File address on FARW is unheld. Any service requesting for hold on that file is serviced.
- Register R14 and register R15 contents are unknown.
- All other registers are preserved over this macro call.

C:

Void

Exception:

CTL- 24	Attempt to unhold address that is not held.
CTL 2D	Zero File Address.

Programming Consideration:

- Used in conjunction with FILEC/FILNC.

Example:

Assembler:

```
UNFRC D7
```

Files address on Level 7 is unheld.

C:

```
unfrc(D7)
```

Files address on Level 7 is unheld.

FILUC – File and Unhold a File Record

FILUC performs the functions of FILEC and UNFRC in a single macro call.

Syntax:

Assembler:

FILUC Dn

Dn	Specifies the data level (D0-DF) on which the held file exists.

C:

void filuc(enum t_lvl level);

level	One of 16 possible values representing a valid data level from the enumeration type t_lvl, expressed as Dx, where x represents the hexadecimal number of the level (0–F). The working storage block on this data level is the record to be filed.

Other variants in C:

void	filuc_ext(enum t_lvl level,unsigned in text);

Entry Conditions:

- The file must be present with hold in the specified level.
- A file address, RID and RCC must be contained in the FARW for the specified data level.

Return Conditions:

Assembler:

- The specified data level will no longer hold a core block.
- CE1HLD is decremented by 1.
- FARW is unchanged and its address is unheld. Any service requesting for hold on that file is serviced.
- Register R14, R15 contents are unknown and other registers are preserved across this macro call.

Exception:

CTL- 23	No block in CBRW.
CTL- 24	Attempt to unhold address that is not held.
CTL- 2D	Zero File Address.

CTL- 2E	RID check failure.
CTL- 2F	RCC check failure.
CTL- 30	Unable to file the record in DASD.
CTL- 32	File block and Storage block size mismatch.

Programming Consideration:

- Program stamp updated on File header.
- RID on FARW cannot be Non-zero. RCC can be zero.
- Preferred to use FILUC instead of FILEC and UNFRC.

Example:

Assembler:

```
FILUC D2
```

Files the record in data level 2 and releases the level.

C:

```
filuc(D6);
```

Files the record in data level 6 and releases the level.

GETFC – Get a Pool file address and storage block.

Used to obtain a pool file address based on the attributes of the specified record ID. It optionally fetches a core block as well.

Syntax:

Assembler:

GETFC Dn, P, O, ID=sd, BLOCK=NO or YES , FILL=xx

Dn	Specifies the data level (D0-DF) on which the file address is to be obtained.
P/O	Parameter that indicates whether a prime (P) or an overflow (O).
ID	***sd***, self-defined term that represents a valid record ID that is defined in at RIAT.
BLOCK	YES/NO, to specify a optional core block also is required.
FILL	Specified if the core block is to be initialized with a specific hexadecimal value.

C:

unsigned int getfc(enum t_lvl level, int type, const char *id, int block, int error, ...);

level One of 16 possible values representing a valid entry control block (ECB) data level from the enumeration type *t_lvl*, expressed as D*x*, where *x* represents the hexadecimal number of the level (0–F). This parameter represents an available file address reference word (FARW) location where the file address will be placed.

type Specify a pool type attribute with the defined terms **GETFC_TYPE0** through **GETFC_TYPE9** associated with the **id** in the record ID attribute table (RIAT). The defined terms **GETFC_PRIME** and **GETFC_OVERFLOW** are still supported for migration purposes only. GETFC_PRIME corresponds to GETFC_TYPE0 and GETFC_OVERFLOW corresponds to GETFC_TYPE1.

Id A pointer to a 2-character string bearing the record ID for which this file allocation is to take place. This ID is used to scan the RIAT table for a match to determine record size and pool type.

block Whether or not a working storage block is to be simultaneously obtained whose characteristics are determined by the RIAT table. Code **GETFC_BLOCK** to obtain a block and a file address, or **GETFC_NOBLOCK** to obtain only a file address.

Additionally, when GETFC_BLOCK is specified, you can code one or both of the terms GETFC_COMM and GETFC_FILL. If you code more than one term for block, you must separate them with a logical OR (|) sign. (See the example that follows.) GETFC_COMM indicates that a common block is to be acquired for the GETFC_BLOCK option. GETFC_FILL along with GETFC_BLOCK indicates that the block acquired should be initialized with the fill character, **FILLC**, specified as the first optional parameter. GETFC_NOCOMM and GETFC_NOFILL, meaning not common block and no fill, respectively, are the defaults and need not be coded.

error	Whether or not control is to be returned to the operational program in the event of an error. Code **GETFC_SERRC** to transfer control to the system error routine (with exit) if the file or core storage cannot be obtained, as requested. Code **GETFC_NOSERRC** to have control returned to the caller.
	The character used to initialize the block if **GETFC_BLOCK** and **GETFC_FILL** are both specified. (The fill character may be specified in hexadecimal.)

Other variants in C: None

Entry Conditions:

- Core block should NOT be present in specified level if BLOCK=YES.

Return Conditions:

Assembler:

- If no error, then Pool file address assigned is placed in the FARW of specified level.
- If BLOCK=YES was specified, core block address is placed in CBRW of specified level.
- Register R14 and register R15 contents are unknown unless BLOCK=YES, in that case R14 has address of the core block.
- All other registers are preserved over this macro call.

C:

- Unsigned integer value representing a file address, or an 8-byte file address defined as type TPF_FA8.

Exception:

CTL 6E4	Record ID not present in RIAT. Invalid RIAT attributes for the Record ID.

Programming Consideration:

- Ensure all pool file addresses received are released.
- Usage of FILL parameter uses more CPU resources.

Example:

Assembler:

`GETFC D1,ID=C'OM',BLOCK=YES` *Gets a prime file address and assigns core as Defined in RIAT for record ID 'OM' on level 1.*

`GETFC D2,O,ID=C'AB'` *Gets an overflow file address as per definition in RIAT*

For Record ID X'C1C2' on level 2.

`GETFC D3,P,ID=C'AB',FILL=00` *Gets an prime file address as per definition in RIAT For Record ID X'C1C2' on level 3 and initialized with zeros.*

C:

```
If(getfc(D2,GETFC_TYPE1,"OM",GETFC_BLOCK|GETFC_FILL,
        GETFC_NOSERRC,' '))
{
```

Gets a file address and assigns Core as defined in RIAT for record Id 'OM' on level 2 and initialize It to blanks.

```
}
```

```
getfc(D1,GETFC_TYPE0,"3P",GETFC_BLOCK,GETFC_NOSERRC,0x00);
 {
```

Gets a file address and assigns Core as defined in RIAT for record Id '3P' on level 1 and initialize It to hex zeros.

```
 }
```

RELFC – Release a Pool file address

Release a pool file address back to the system.

Syntax:

Assembler:

RELFC Dn

Dn	Specifies the data level (D0-DF) on which the file address is present.

C:

void relfc(enum t_lvl level);

level	One of 16 possible values representing a valid entry control block (ECB) data level from enumeration type *t_lvl*, expressed as D*x*, where *x* represents the hexadecimal number of the level (0–F). This parameter identifies the file address reference word (FARW) containing the pool file address to be returned to the system.

Other variants in C: None

Entry Conditions:

- File address to be released should be present in FARW of data level specified.

Return Conditions:

Assembler:

- Register R14 and register R15 contents are unknown and all other registers are preserved over this macro call.

C:

- Void. Additionally, the CE3RELFRC field in ECB page 3 contains a Condition name CE3_RELFC_NRM having a value of 0 indicates normal return.

Exception:

CTL- F	Double release of pool file address

Programming Consideration:

- Ensure all pool file addresses received are released.

Example:

Assembler:

```
RELFC D1
```

Releases file address on FARW of level 1.

C:

```
relfc(D1)
```

Releases file address on FARW of level 1.

RLCHA – Release Chain

Release a chain of pool files (linked by forward chains) back to the system.

Syntax:

Assembler:

RLCHA

Register R15 Should point to 12 byte data formatted as below:

Bytes 0-1	Record ID
Byte 2	Record code check
Bytes 3-7	Not used
Bytes 8-11	File address of first to be released

C:

void rlcha(struct stdhdr *hdr);

hdr This argument is a pointer to struct stdhdr, see tpfapi.h which describe the z/TPF standard record header.

Other variants in C: None

Entry Conditions: None

Return Conditions:

Assembler:

- Contents of register R14 are F'12' and register R15 is unknown and all other registers are preserved over this macro call.

C: void

Exception:

CTL- F	Double release of pool file address
CTL FFFFFC	Error while trying to release chain records with remaining not released.

Programming Consideration:

- Ensure all succeeding record have same RID and RCC. Else the RLCHA exits.
- This macro creates new ECBs to handle the release which may deplete storage.

Example:

Assembler:

`RLCHA` *Releases chain of pool files based on parameter pointed by R15.*

C:

```
cp0hdr = ecbptr()->ce1cr5;
rlcha(cp0hdr);
```

Releases chain of pool files pointed at cp0hdr.

5.6 zTPF Stream File system support

- z/TPF C language support is a hosted implementation that conforms to ANSI and ISO standard C as defined in ANSI/ISO 9899-1990. A hosted implementation supports:
    - The complete C language specification
    - The main function invoked at program startup
    - All of the functions, macros, type definitions, and objects defined in the standard C library.

- Streams and Files are supported in the z/TPF C run-time library. z/TPF file system C functions operate through the z/TPF stream I/O support

- The z/TPF system supports stream input/output (I/O) at an abstract level, with a hierarchical file system modeled on the UNIX and Portable Operating System Interface for Computer Environments1 standards (POSIX.1)

- File names are specified as either:
    - Absolute path names, which begin with a leading / (slash, 0x61)), and traverse a directory path starting from the root directory ("/"); for example: "/etc/bin/ls".
    - Relative path names, which do not have a leading /, but begin with a file, subdirectory, or symbolic link name that is contained in the current working directory (cwd); for example: "my.subdir/my.file".

- The z/TPF file system APIs work at two levels:
    - Buffered I/O
    - System-level I/O.

- The more abstract level is the **buffered I/O functions**, declared in <stdio.h>. The buffered I/O functions operate on a pointer to FILE. Examples include fopen, fclose and fflush; fread and fwrite; the formatted I/O functions such as fprintf and fscanf; the character I/O functions such as fgetc, fgets, fputc, fputs; and many others.

- The **system-level I/O** functions, declared in <unistd.h>, <fcntl.h>, <dir.h>, and other headers, are appropriate in cases where an application needs to work on the directory structure itself or where more control over files than is provided by the buffered I/O functions is required. The system-level I/O functions operate on a file descriptor, a path name (which is a C string), or a pointer to DIR. Examples include creat, open, close, read, write, fcntl, chdir, chown, chmod, mkdir, mknod, and stat, and many others.

ZFILE Commands

- The ZFILE commands are a set of commands that allow you to work with and manage files on the file system.

- Online help information is available for this command. To display the help information, enter one of the following:

  - ZFILE HELP cd
  - ZFILE HELP
  - ZFILE ?

- You can redirect the standard input (stdin) stream from the keyboard to a file by specifying the redirection character (<) followed by the file name that you want the input read from.

- You can redirect the standard output (stdout) stream from the display terminal to a file by specifying one of the redirection characters (> or >>) followed by the file name that you want the output written to. The > character writes the output to a file. The >> character appends the output to an existing file.

- You can redirect the standard error (stderr) stream from the display terminal to a file by specifying one of the redirection characters (2> or 2>>) followed by the file name that you want the error output written to. The 2> character writes the error output to a file. The 2>> character appends the error output to an existing file.

- You can use a vertical bar, or pipe (|), to direct data so that the output from one process becomes the input to another process. This type of one-way communication allows you to combine ZFILE commands on one line to create a pipeline. For example, the following pipeline uses the standard output (stdout) stream from the ZFILE ls command and redirects it to the standard input (stdin) stream of the ZFILE grep command to search for those lines containing the word Jan

  - `ZFILE ls -l | grep Jan`

FOPEN – Open a file

This function opens a file.

Syntax:

Assembler: None

C:

FILE *fopen(const char *filename, const char *mode);

Filename	The name of the file to be opened as a stream.
Mode	The access mode for the stream.

Other variants in C: None

Entry Conditions:

- The z/TPF file system does not use different file formats for binary and text files. Each line of a text file ends with an EBCDIC '\n' (0x15) character. Both text and binary files are treated as streams of bytes; there is no structure imposed on or conversion performed on the byte stream.

Attention: Use the w, w+, wb, w+b, and wb+ parameters with care; data in existing files of the same name will be lost.

- *Text files* contain printable characters and control characters organized into lines. Each line ends with a new-line character.
- *Binary files* contain a series of characters. Any additional interpretation or structure imposed on a binary file is the responsibility of the application.
- When you open a file with a, a+, ab, a+b, or ab+ mode, all write operations take place at the end of the file. Although you can reposition the file pointer using the fseek, fsetpos, or rewind function, the write functions move the file pointer back to the end of the file before they carry out any output operation. This action prevents you from overwriting existing data.
- When you specify the update mode (using + in the second or third position), you can read from and write to the file. However, when switching between reading and writing, you must include an intervening positioning function such as fseek, fsetpos, rewind, or fflush. Output may immediately follow input if the end-of file (EOF) was detected.

Return Conditions:

Assembler: None

C:

- If successful, the fopen function returns a pointer to the object controlling the associated stream.

- A NULL pointer return value indicates an error.

Exception: None

Programming Consideration:

- The z/TPF collection support file system does not support creating, updating, or deleting files in 1052 or UTIL state. Special files may or may not be writable in 1052 or UTIL state depending on the device driver implementation.
- If the TPF_REGFILE_RECORD_ID environment variable is set to a 2-character string when a new regular file is created in the z/TPF collection support file system (TFS), its value is used as the record ID for all pool file records that are allocated to store the contents of the file.

Example:

Assembler: None

C:

The following example attempts to open a file for reading.

```
#include <stdio.h>
int main(void)
{
   FILE *stream;

   /* The following call opens a text file for reading */

   if ((stream = fopen("myfile.dat", "r")) == NULL)
      printf("Could not open data file for reading\n");
}
```

FREOPEN – Redirect an open file

This function redirects an open file.

Syntax:

Assembler: None

C:

FILE *freopen(const char *filename, const char *mode, FILE *stream);

Filename	The name of the file to replace an open stream.
Mode	The access mode for the reopened stream.
Stream	The stream to be reopened on the new file name.

Other variants in C: None

Entry Conditions:

- This function closes the file currently associated with **stream** and pointed to by **stream**, opens the file specified by **filename**, and then associates the stream with it.
- The freopen function opens the new file with the type of access requested by the **mode** argument.
- The **mode** argument is used as in the fopen function. You can also use the freopen function to redirect the standard stream files stdin, stdout, and stderr to files that you specify.
- The file pointer parameter to the freopen function must point to a valid open file. If the file has been closed, the behavior is not defined.

Return Conditions:

Assembler: None

C:

- If successful, the freopen function returns the value of **stream**, the same value that was passed to it, and clears both the error and EOF indicators associated with the stream.
- A failed attempt to close the original file is ignored.
- If an error occurs when reopening the requested file, the freopen function closes the original file and returns a NULL pointer value.

Exception: None

Programming Consideration:

- The z/TPF collection support file system does not support creating, updating, or deleting files in 1052 or UTIL state. Special files may or may not be writable in 1052 or UTIL state depending on the device driver implementation.

Example:

Assembler: None

C:

The following example closes the ***stream*** *data stream and reassigns its stream pointer*:

```
#include <stdio.h>
int main(void)
{
   FILE *stream, *stream2;
   stream = fopen("myfile.dat","r");
   stream2 = freopen("myfile2.dat", "w+", stream);
}
```

FSCANF – Read and format data

This function read and format data.

Syntax:

Assembler: None

C:

int fscanf (FILE *:stream, const char *format, …);

Stream	The stream from which the text input will be read.
Format	A specification of the format and conversions to be applied to the input text.

Other variants in C: None

Entry Conditions:

- Specified file should exist to perform scanning

Return Conditions:

Assembler: None

C:

- fscanf return the number of input items that were successfully matched and assigned. The return value does not include conversions that were performed but not assigned (for example, suppressed assignments).

- The functions return EOF if there is an input failure before any conversion or if EOF is reached before any conversion. Therefore, a return value of 0 means that no fields were assigned; there was a matching failure before any conversion. Also, if there is an input failure, the file error indicator is set, which is not the case for a matching failure.

- The ferror and feof functions are used to distinguish between a read error and an EOF.

  Note: EOF is reached only when an attempt is made to read beyond the last byte of data.

- Reading up to and including the last byte of data does ***not*** turn on the EOF indicator.

Exception: None

Programming Consideration: None

Example:

Assembler: None

C:

The following example opens the file myfile.dat for reading and then scans this file for a string, a long integer value, a character, and a floating-point value.

```
#include <stdio.h>
#define  MAX_LEN  80

int main(void)
{
   FILE *stream;
   long l;
   float fp;
   char s[MAX_LEN + 1];
   char c;

   stream = fopen("myfile.dat", "r");

   /* Put in various data. */
   fscanf(stream, "%s", &s[0]);
   fscanf(stream, "%ld", &l);
   fscanf(stream, "%c", &c);
   fscanf(stream, "%f", &fp);

   printf("string = %s\n", s);
   printf("long double = %ld\n", l);
   printf("char = %c\n", c);
   printf("float = %f\n", fp);
}
```

FPRINTF – Format and write data

This function formats and writes data to a stream.

Syntax:

Assembler: None

C:

int fprintf(FILE *stream, const char *format, …);

Stream	The stream to be written.
Format	A specification of the output text format

Other variants in C: None

Entry Conditions:

- Specified file should exist in order to write data to the file

Return Conditions:

Assembler: None

C:

- If successful the fprintf function return the number of characters written. The ending null character is not counted.
- A negative value indicates that an output error has occurred.

Exception: None

Programming Consideration:

- The z/TPF collection support file system does not support creating, updating, or deleting files in 1052 or UTIL state. Special files may or may not be writable in 1052 or UTIL state depending on the device driver implementation.

Example:

Assembler: None

C:

The following example opens the file myfile.dat for writing

```
#include <stdio.h>
int main(void)
{
   FILE *stream;
   char *s=”Test of fprintf”;
   stream = fopen("myfile.dat", "w");
   fprintf(stream,"string = %s\n", s);
}
```

FSEEK – Change file position

This function sets a file position.

Syntax:

Assembler: None

C:

int fseek(FILE *stream, long int offset, int origin);

Stream	The stream for which the current position will be set.
Offset	The distance in bytes from the origin.
Origin	A position in the stream. The **origin** must be one of the following constants defined in the `stdio.h` header file: **SEEK_SET** Beginning of the file. **SEEK_CUR** Current position of the file pointer. **SEEK_END** End of file.

Other variants in C: None

Entry Conditions:

- Attempting to reposition before the start of the file causes the fseek function to fail.

Return Conditions:

Assembler: None

C:

- If it successfully moves the pointer, the fseek function returns a zero value.
- A nonzero return value indicates an error. On devices that cannot seek, such as terminals and printers, the return value is nonzero.

Exception: None

Programming Consideration: None

Example:

Assembler:

C:

The following example opens a myfile.dat file for reading. After performing input operations (not shown), the file pointer is moved to the beginning of the file.

```
#include <stdio.h>
int main(void)
{
   FILE *stream;
   int result;

   if (stream = fopen("myfile.dat", "r"))
   { /* successful */

   /* moves pointer to the beginning of the file */
   if (fseek(stream, 0L, SEEK_SET));
   { /* if not equal to 0, then error ... */
   }
   else
   {
    /* fseek() successful  */
   }
}
```

FSETPOS – Set file position

This function sets a file position.

Syntax:

Assembler: None

C:

int fsetpos(FILE *stream, const fpos_t *pos);

Stream	The stream for which the current file position will be set.
Pos	The new file position for the stream.

Other variants in C: None

Entry Conditions:

- Specified file should exist before calling fsetpos function.

Return Conditions:

Assembler: None

C:

- If the fsetpos function successfully changes the current position of the file, it returns a zero value.
- If there is an error, errno is set and a nonzero value is returned.

Exception: None

Programming Consideration: None

Example:

Assembler:

C:

The following example opens a file called myfile.dat for reading. After performing input operations (not shown), the file pointer is moved to the beginning of the file and reads the first byte again.

```
#include <stdio.h>
int main(void)
{
   FILE *stream;
   int retcode;
   fpos_t pos, pos1, pos2, pos3;
   /* existing file 'myfile.dat' has 20 byte records */
   char ptr[20];

   /* Open file, get position of file pointer,
     and read first record */

   stream = fopen("myfile.dat", "rb");
   fgetpos(stream,&pos);
   pos1 = pos;
   if (!fread(ptr,sizeof(ptr),1,stream))
       printf("fread error\n");

   /* Perform a number of read operations.
     The value of 'pos' changes if 'pos' is
     passed to fgetpos() */
⋮
   /* Re-set pointer to start of file and
     re-read first record  */
   fsetpos(stream,&pos1);
   if (!fread(ptr,sizeof(ptr),1,stream))
       printf("fread error\n");

   fclose(stream);
}
```

FCLOSE – Close file

This function closes a file.

Syntax:

Assembler: None

C:

int fclose(FILE* *stream*);

Stream	The name of the file to be closed as a stream.

Other variants in C: None

Entry Conditions:

- Flushes a stream, and then closes the file associated with that stream. Afterwards, the function releases any buffers associated with the stream. To *flush* means that unwritten buffered data is written to the file, and unread buffered data is discarded.
- A pointer to a closed file *cannot* be used as an input value to the freopen() function.

Note:

- The storage pointed to by the FILE pointer is freed by the fclose() function. An attempt to use the FILE pointer to a closed file is not valid. This restriction is true even when fclose() fails.
- If an application has locked a (FILE *) object (with flockfile() or ftrylockfile()), it is responsible for relinquishing the locked (FILE *) object (with funlockfile()) before calling fclose(). Failure to relinquish a locked (FILE *) object may cause deadlock (or looping).

Return Conditions:

Assembler: None

C:

- If successful closing the stream, fclose() returns 0.
- If a failure occurs in flushing buffers or in outputting data, fclose() returns EOF. An attempt will still be made to close the file.

Exception: None

Programming Consideration: None

Example:

Assembler:

C:

```
/* This example opens a file myfile.dat for reading as a
stream and then closes the file.*/
```

```
#include <stdio.h>
int main(void)
{
   FILE *stream;

   stream = fopen("myfile.dat", "r");

  .
  .
  .

   if (fclose(stream))   /* Close the stream. */
   {
      printf("fclose error\n");
   }
}
```

Q1. How to retrieve fix and pool file?

Q2. Can we create the fix file while processing the entry?

Q3. After RELFC what will happen with fix and pool file?

Q4. What is the lifespan of short term pool file?

Q5. Why WAITC is needed after find macros?

Q6. How can we avoid deadlock condition in the system?

Q7. After RELFC core block and file address both will release?

Q8. How can we check whether FACE is successful or not?

6. Process Management

6.1 CPU Loop (Dispatching)

- The process of selecting an entry for execution is called Scheduling.

- In zTPF, a system program called the **CPU loop** is responsible for scheduling an entry for execution by an I-stream.

- The order of processing priority is determined by the sequence in which the CPU loop interrogates queues that identify work items to be dispatched.

- The term **list** refers to the CPU loop queues.

- The **CPU loop lists**, in order of processing priority, are:

  - **Cross list:** The cross list is used to dispatch entries between I-stream engines.
  - **Time dispatch list**: The time dispatch list is used to dispatch time-sensitive events.
  - **Ready list:** The ready list is used to start processing a high-priority task, or to return control to an entry that was already created as the result of some completed system activity.
  - **Input list and OSA input message list:** The input list is used to dispatch a new entry for processing an input message that has an I-stream affinity to this particular I-stream engine.
    The OSA input message list is a shared list that is used to process input messages that arrive on the network from an Open Systems Adapter (OSA)-Express connection.
  - **Deferred list:** The deferred list is used to delay the execution of an entry that has an I-stream affinity to this particular I-stream engine.

- There are two I-Stream unique secondary lists in addition to the main CPU loop lists. These two lists are:

  - **Virtual file access count(VCT) list:** All items on the VCT list will be processed every pass through all the items on the input list.
  - **Suspend list:** One item on the suspend list is processed every pass through all items on the input list.

- The **ZACVL** command is used to change the maximum number of times that the input list is serviced before the deferred list is serviced.

6.2 Operation zero program(OPZERO)

- OPZERO refers to a collection of system programs associated with communications control in the z/TPF system.

- OPZERO creates an entry control block (ECB) and associates the input message with the ECB

- OPZERO then passes control (and the ECB) to COMM SOURCE (Communications Source Program) to continue input message processing

6.3 Message Processing

- An input message causes an Entry to be created and application program segments to be dispatched for processing the input message.

- The Entry makes requests of the control program for system services such as:
    - Input and output (I/O)
    - Storage allocation (both file and main storage)
    - Program fetches (called Enter/Back in the z/TPF system)
    - Release of control.

- Input messages received from the communication facilities are ultimately placed in main storage buffers. As a result, the I/O interrupt handler invokes system programs and puts work items (associated with the input messages) on the processing **queues (lists).**

- A **dispatch control list (DCL) or CPU loop list** is formally a queue or waiting line with a top and bottom. Work is always dispatched (removed) from the top of a list and the succeeding items (more work) are promoted from within the list.

- The CPU loop inspects the cross list, ready list, input list, and deferred list, and checks status indicators

- When the CPU loop finds an item on the **input list, control is given to OPZERO**. OPZERO creates an ECB and gives control to COMM SOURCE, which selects the application program segment necessary to begin interpreting the input message.

- As SVC interrupts are received, the requests for system services are processed. Often this involves I / O.

- When the request for I/O is serviced:
  - Put the I/O request on an I/O device queue.
  - If the control program finds that the application currently in control must wait for the activity associated with the requested system service to complete, the Entry is suspended and the CPU loop receives control again to interrogate the lists and check for the arrival of more input messages
- As I/O completes, work items are placed on the **ready list**.
- The CPU Loop can:
  - Return to a previously suspended Entry, which is an item on the ready list
  - Start a new application (Entry) as the result of an item on the input list
  - Continue looping until input messages arrive.
- The application, in general, refers to a collection of programs that interprets the semantics of the entry, process it and generate a response.
- A TPF program can invoke another program by using one of the ENTER or CREATE macro services.
- An ECB is granted **500-millseconds** of processing time. If it exceeds the time limit, a system error (CTL-10) is issued and the ECB is aborted.
- However, the time limit is reset when the ECB has to give up control to wait for an I/O to be completed or it has been deferred from execution.

6.4 Program Linkage - Enter/Back

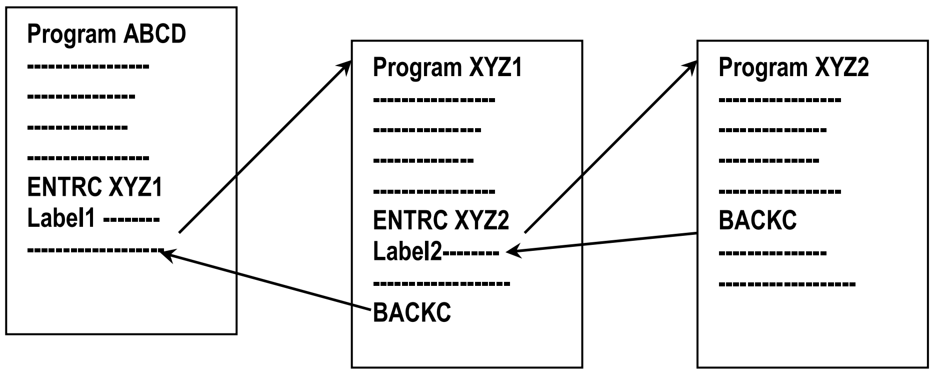
- A TPF program can invoke another by using one of the ENTER type macros listed below:
- ENTRC – Enter with return. The calling program expects the control to return back. The BACKC macro returns the control to the last program that issued an ENTRC.
- ENTNC – Enter with no return. The calling program does not expect the control to return back.
- ENTDC – Enter and drop previous programs. The Enter-Back macro control information that was saved is reinitialized to remove linkages to all previous programs..

- SWISC TYPE=ENTER — Transfer the ECB to another I-stream engine and drop previous programs. This macro performs the function of ENTDC while transferring the ECB to another I-stream engine.

- Here cross list will come into picture as this macro will switch the entry to available I-Stream.

ENTRC

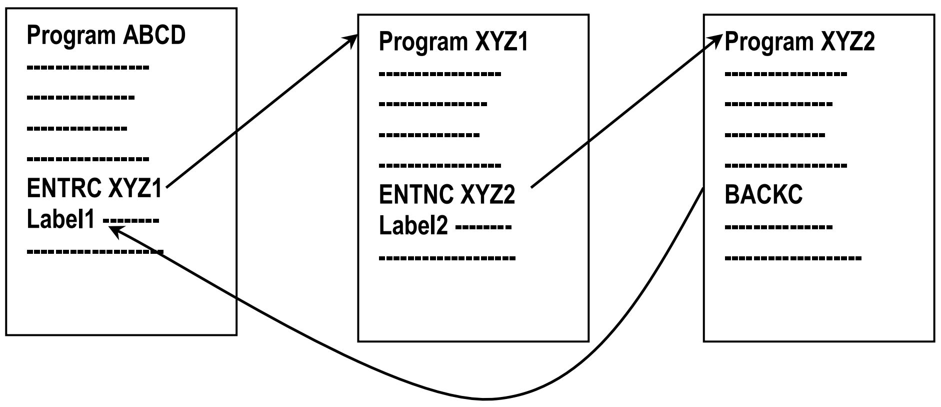
When a program issues a return call, control is transferred to the called program. The control is returned back to the calling program on BACKC.



None	ABCD	ABCD,XYZ1	ABCD,XYZ1,XYZ2 (Nesting on entry)
None	None	ABCD	ABCD, XYZ1 (Nesting on return)

ENTNC

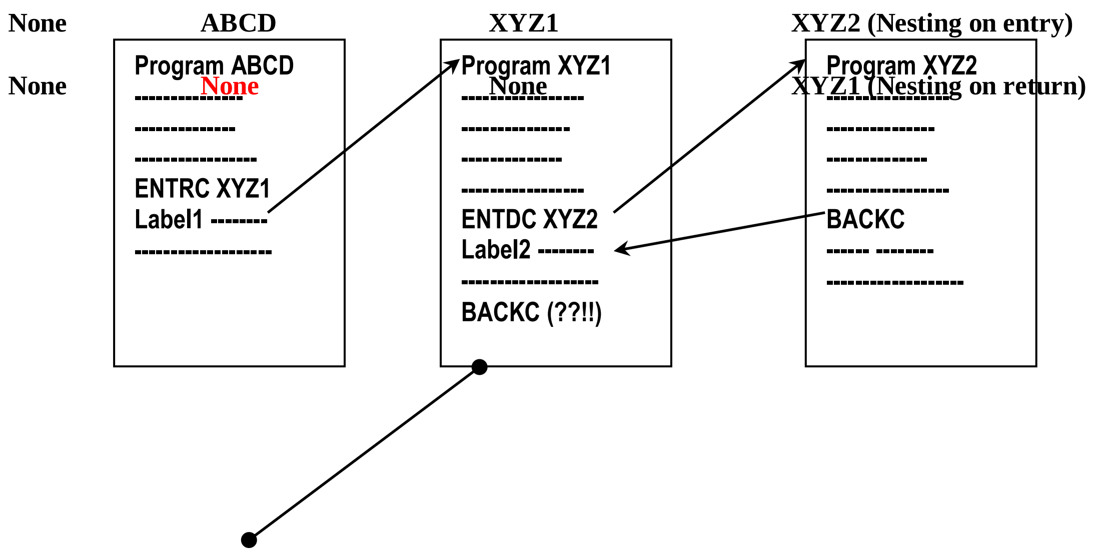
When a program issues a no return call, the control is transferred to the called program. The ECB doesn't come back to the calling program.



None	ABCD	ABCD	ABCD (Nesting on entry)
None	None	-NOCONTROL-	ABCD (Nesting on return)

ENTDC

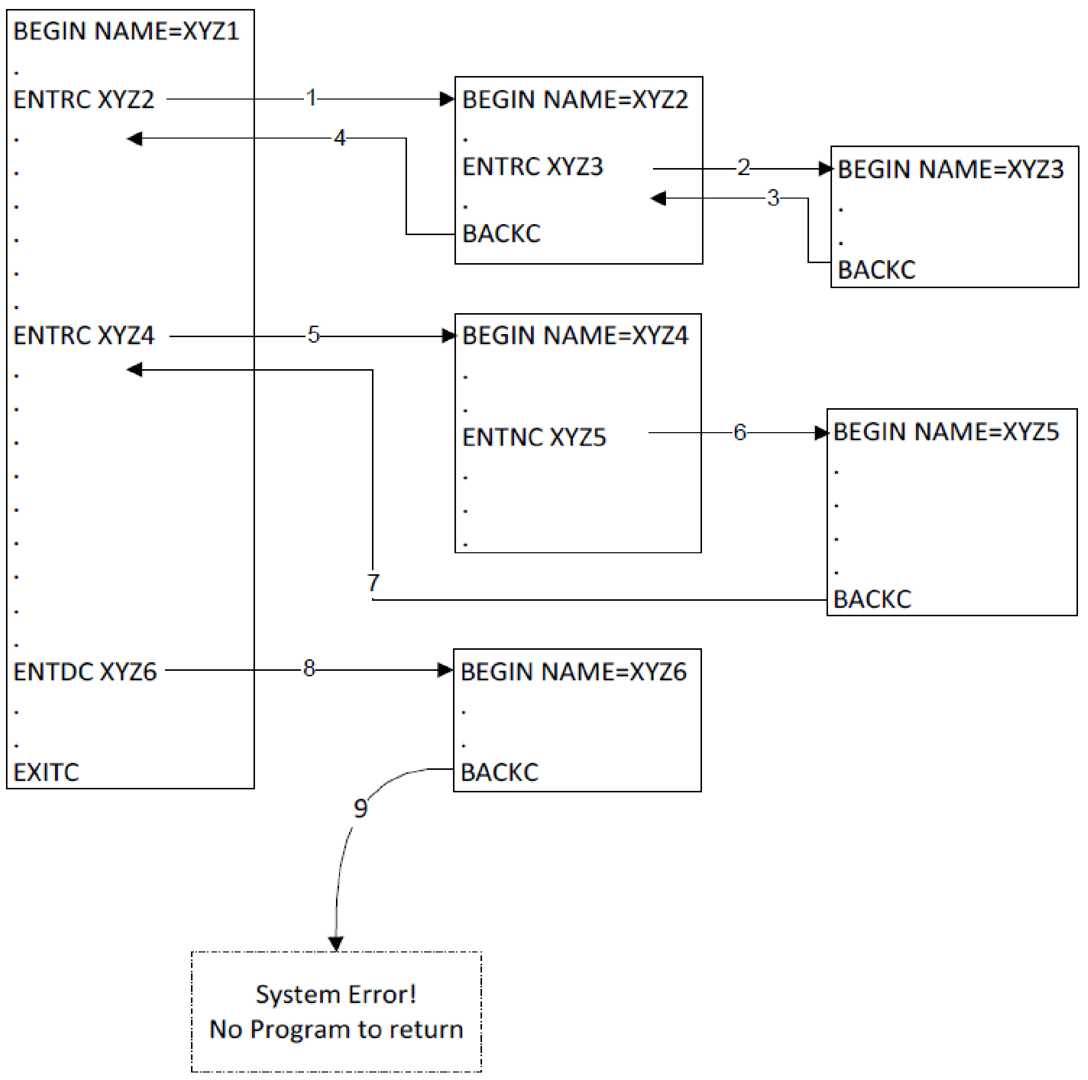
This call forces the control to be transferred to the called program, like the no return call. In addition, the program nesting area is cleared.



Point of no-return!!! (ECB dumps and exits)

6.5 Program Nesting

- Several related programs can be used to process a single message under control of a single ECB.
- A chain of ENTRC calls, all under the control of same ECB, is called nesting.
- As shown in the ECB format, a program nesting area is maintained within the ECB to hold Enter/Back linkage.
- When an application program issues the ENTDC macro, all items in the program nesting area are cleared.
- The following diagram illustrates the program nesting concept with various combination of Enter/Back macros.



6.6 Create entries with Create Macro

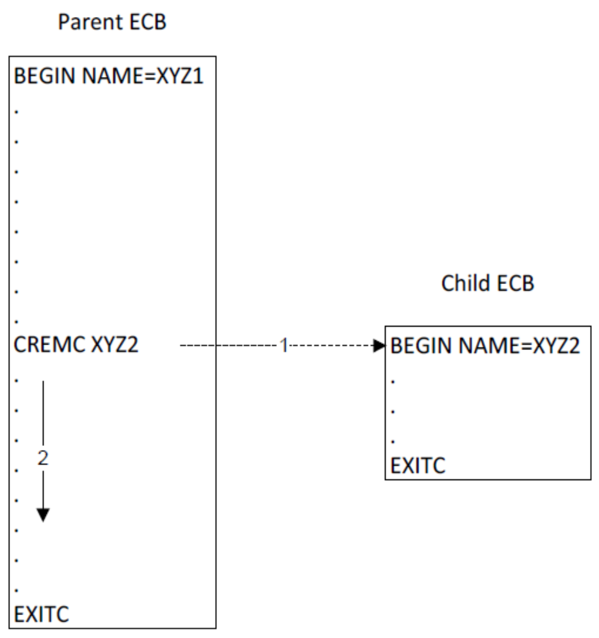
- A set of macros permits an active Entry to create another independent Entry.

- The creating Entry can pass information to the created Entry. The procedure is similar to a control transfer but is done on behalf of an existing ECB-controlled program rather than on behalf of a control program component.

- The different types of create macro requests are:

  - Create an immediate Entry (CREMC), which places the Entry on the ready list.
  - Create new synchronous ECBs (CRESC) which places the Entry on the ready list.

- Create time-initiated Entry (CRETC), which places the Entry on the ready list after a specified time interval; an attached storage block can be passed to the new Entry.
- Create a low priority deferred Entry (CREXC), which places the Entry on the deferred list for low-priority deferred processing.
- Create an immediate or deferred Entry with attached storage block (CREEC), which places the Entry on the ready list or the deferred list. When the new ECB is created, information from the creating ECB is placed in a storage block attached to the new ECB.
- Create an Entry on a specified I-stream engine (SWISC TYPE=CREATE), which places the Entry on the input, ready, or deferred list on a designated I-stream engine.



6.7 Process Management Macros

ENTRC – Enter Program with Return

ENTRC transfers control to the specified program. The address of NSI in the calling program is saved for the expected return.

Syntax:

Assembler:

ENTRC Program

Program	Name of the program to be entered.

C:

void *entrc(const char *program,…);

program	For *entrc*, a pointer to the application program to be called. *program* must be the 4-character name of a basic assembler language (BAL) segment name, a TPF segment name or a TPF ISO-C DLM name.

Other variants in C: None**.**

Entry Conditions: None

Return Conditions:

Assembler:

- Control resumes from the NSI of the caller when the called program issues a BACKC.
- The condition code and registers R14 and R15 are unpredictable and other registers as modified by the called segment is returned.

C:

- The CBRW has been modified to indicate that no block is held.

Exception:

CTL- 60	Maximum nesting level exceeded.
CTL- 63	Program not found or Program header corrupted.

Programming Consideration:

- It is advisable to store all registers used as base in calling program and restored on return
- Code ENTRC if isolated or independent task to perform like validation utility or DB update utility.

Example:

Assembler:

```
ENTRC ABCD
```
Enters program ABCD and returns to NSI.

C:

```
void call_programs(struct TPF_regs *regs)
{
entrc("ASM0",NULL);
return;
}
```
Enters program ASM0 and returns to called program.

ENTNC – Enter Program with No Return

This general macro transfers control to the specified operational program. The entering program stops.

Syntax:

Assembler:

ENTNC Program

Program	Name of the program to be entered.

C: None

Entry Conditions: None

Return Conditions:

Assembler:

- Control is not returned to caller program.

Exception:

CTL- 63	Program not found or Program header corrupted.

Programming Consideration:

- This macro is used when no return is expected by caller from the called program.
- Usually used during completion of an entry process.

Example:

Assembler:

```
ENTNC ABCD
```

Enters program ABCD with No return.

ENTDC – Enter Program and Drop Previous Programs

ENTDC transfers control to the specified program and all the program references in the program nesting are purged.

Syntax:

Assembler:

ENTDC Program

Program	Name of the program to be entered.

C:

void entdc(void (*segname)(), struct TPF_regs *regs);

segname	For *entdc*, a pointer to the external macro to be called. *segname* must map to a BAL segment name, a TPF segment name or a TPF ISO-C DLM name.
regs	Treated as a pointer to *struct TPF_regs* that contains 8 signed long integer members, r0-r7. These members are used to load general registers R0-R7 before passing control. If this argument is coded as NULL, no registers are loaded.

Other variants in C:

void	_ENTDC(const char *segname, struct TPF_regs *regs);

Entry Conditions: None

Return Conditions:

Assembler:

- Control is NOT returned to caller program.

C:

- The CBRW has been modified to indicate that no block is held.

Exception:

CTL- 63	Program not found or Program header corrupted.

Programming Consideration:

- Control is transferred to the system error subroutine when an ENTDC macro is issued between the executions of an ENTRC macro and the BACKC macro.

Example:

Assembler:

```
      ENTDC ABCD
```

Enters program ABCD with No return.

C:

```
void ASM0();
struct TPF_regs *regs=(struct TPF_regs *)&(ecbptr()->ebx000);
.
.
.
entdc(ASM0, regs);
```

Enters program ASM0 without return along with the values in the indicated registers.

BACKC – Return to Previous Program

BACKC transfers control to the last program that performed an ENTRC.

Syntax:

Assembler:

BACKC

C: None

Entry Conditions: None

Return Conditions:

Assembler:

- Control is not returned to caller program issuing the BACKC.

Exception:

CTL- 61	No program nested. BACKC cannot find a program to return control.

Programming Consideration:

- The program that last issued the ENTRC macro will regain control.
- The registers R0-R7 have the same value they had when the BACKC macro was issued.
- The condition code and contents of R14 and R15 are unpredictable.

Example:

Assembler:

```
BACKC                    Returns control to program that last did a ENTRC.
```

DLAYC – Delay Processing

DLAYC delays processing of the ECB by placing it in the end of Input List.

Syntax:

Assembler:

DLAYC

C:

void dlayc(void);

Other variants in C:

None.

Entry Conditions: None

Return Conditions:

Assembler:

- Control is returned to NSI when processing resumes.
- Register R14, R15 contents are unknown and other registers are preserved across this macro call.

Exception: None

Programming Consideration:

- Records should not be held by ECB when DLAYC is executed.
- When macro runs, the 500 ms program timeout is reset.
- Should be used optimally as frequent usage might deplete resources.

Example:

Assembler:

```
DLAYC
```
Pause for a while.

C:

```
dlayc();
```
It suspends processing the current program and places the ECB in input list.

DEFRC – Defer Processing of Current Entry

DEFRC is used to defer the processing of an entry until the amount of activity in the system is sufficiently low to allow for completion of this low priority task. The entry is placed on the Defer list, but continues to retain all the core blocks assigned.

Syntax:

Assembler:

DEFRC

C:

void defrc(void);

Other variants in C:

None

Entry Conditions:

- ECB should not be holding any files.

Return Conditions:

Assembler:

- Control is returned to NSI when processing resumes.
- Register R14, R15 contents are unknown and other registers are preserved across this macro call.

Exception:

CTL- D	Defer issued holding a record.

Programming Consideration:

- When macro runs, the 500 ms program timeout is reset.

Example:

Assembler:

```
DEFRC
```
Pause for a while.

C:

`defrc();` *It suspends processing the current program and places the ECB in deferred list*.

EXITC – Processing of an Entry is Complete

EXITC exits processing by ending the life of the entry by releasing the ECB, any working storage, or program record blocks held the ECB.

Syntax:

Assembler:

EXITC

C:

void exit(int return_code);

return_code	An indicator value indicating status.

Other variants in C:

void	abort(void);

Entry Conditions:

- All held resources and records must be released before issuing an EXITC.

Return Conditions:

Assembler:

- ECB and related resources will be released back to the system.

Exception:

CTL- 8	Holding a record during exit.
CTL-13	Tape open at exit time.

Programming Consideration: None

Example:

Assembler:

```
EXITC
```
Exit processing of Entry.

C:

```
exit(0);
```
Exits the ECB when processing is completed.

CREDC – Create a Deferred Entry

CREDC is used to create an independent ECB for deferred processing. Parameters of maximum 104 bytes can be passed to the new ECB.

CP moves the parameters passed into an interim block of working storage (SWB) and adds this block to the deferred list. OPZERO initializes the ECB with the parameters in its work area (EBW000), then releases the interim block and executes an ENTNC to the specified program.

Syntax:

Assembler:

CREDC Program

Program Name of the program that is to be activated with the created ECB.

C:

void credc(int length const void *parm, void (*segname)());

length	An integer containing the number of bytes to be passed to the created ECB. A value of zero indicates that no parameters will be passed. A maximum of 104 bytes may be passed.
parm	A pointer of type *void* to the parameters to be passed.
segname	For *credc*, a pointer to the external function to be called. *segname* must map to an assembler segment name, a TPF segment name (or transfer vector), or a TPF ISO-C dynamic load module (DLM) name.

Other variants in C:

void	_CREDC(int length, const void *parm, const char *segname);

Entry Conditions:

- R14 must contain the number of bytes of parameters to be passed to the created ECB work area.
- R15 must contain the address of the start of the parameters which are to be passed.

Return Conditions:

Assembler:

- Control is returned to NSI.
- Register R14, R15 contents are unknown and other registers are preserved across this macro call.

Exception:

CTL- 14	Length of the parameter list is more than the maximum length allowed.

Programming Consideration:

- The ECB issuing the CREDC macro may be forced into a WAIT if there is insufficient storage available to buffer the parameters.
- When adequate working storage is available, the CREDC macro is executed and a return is made to the instruction after the macro expansion.

Example:

Assembler:

```
LA R14,10
LA R15,PARMADDR
CREDC XYZ1
```

Creates entry to program XYZ1 with 10 bytes of data pointed by R15.

C:

```
void ASM0();
char *parmstring=”ASM”;
credc(strlen(parmstring),parmstring,ASM0);
```

The above example creates a deferred entry for program ASM0, passing string ASM as input data to the program.

CREXC – Create a Low Priority Deferred Entry

CREXC is similar to CREDC macro but it does extra checking regarding the availability of working storage, before creating the ECB.

Syntax:

Assembler:

CREXC program

Program	Name of the program that is to be activated with the created ECB.

C:

void crexc(int length, const void *param, void (*segname)());

length	An integer containing the number of bytes to be passed to the created ECB. A value of zero indicates that no parameters will be passed. A maximum of 104 bytes may be passed.
parm	A pointer of type *void* to the parameters to be passed.
segname	For *crexc*, a pointer to the external function being called. *segname* must map to an assembler segment name, a TPF segment name or a TPF ISO-C DLM name.

Other variants in C:

void	_CREXC(int length, const void *param, const char *segname);

Entry Conditions:

- R14 must contain the number of bytes of parameters to be passed to the created ECB work area.
- R15 must contain the address of the start of the parameters which are to be passed.

Return Conditions:

Assembler:

- Control is returned to NSI.
- Register R14, R15 contents are unknown and other registers are preserved across this macro call.

Exception:

CTL-14	Length of the parameter list is more than the maximum length allowed.

Programming Consideration:

- Utilize CREXC for low priority maintenance utilities where creation of many entries is common. This will ensure normal system operation without impairment, which could be caused by depletion of resources.

- When adequate working storage is available, the CREDC macro is executed and a return is made to the instruction after the macro expansion.

Example:

Assembler:

```
XR R14,R14
CREXC XYZ2
```

Creates entry to program XYZ2 with no parameters.

C:

```
void ASM0();
char *parmstring=”ASM”;
crexc(strlen(parmstring),parmstring,ASM0);
```

The above example creates a deferred entry for program ASM0,passing the string ASM as input data to the program.

CREMC – Create a New ECB for Immediate Entry

CREMC is used to create an independent ECB for immediate processing by the requested program. Parameters of maximum 104 bytes can be passed to the new ECB.

Syntax:

Assembler:

CREMC Program

Program	Name of the program that is to be activated with the created ECB.

C:

void cremc(int length, const void *parm, void (*segname)());

length	An integer containing the number of bytes to be passed to the created ECB. A value of zero indicates that no parameters will be passed. A maximum of 104 bytes may be passed.
parm	A pointer of type *void* to the parameters to be passed.
segname	For *cremc*, a pointer to the external function being called. *segname* must map to an assembler segment name, a TPF segment name or a TPF ISO-C DLM name.

Other variants in C:

void	_CREMC(int length, const void *parm, const char *segname);

Entry Conditions:

- R14 must contain the number of bytes of parameters to be passed to the created ECB work area.
- R15 must contain the address of the start of the parameters which are to be passed.

Return Conditions:

Assembler:

- Control is returned to NSI.
- Register R14, R15 contents are unknown and other registers are preserved across this macro call.

Exception:

CTL-14	Length of the parameter list is more than maximum length allowed.

Programming Consideration:

- The ECB issuing the CREDC macro may be forced into a WAIT if there is insufficient storage available to buffer the parameters.
- When adequate working storage is available, the CREMC macro is executed and a return is made to the instruction after the macro expansion.

Example:

Assembler:

```
XR    R14,R14
CREMC XYZ1
```

Creates entry to program XYZ1 with no parameters.

C:

```
void ASM0();
char *parmstring=”ASM”;
cremc(strlen(parmstring),parmstring,ASM0);
```

The above example creates an ECB dispatched from the ready list for program ASM0, passing string ASM as input data to the program.

CREEC – Create a New ECB with Attached Core Block

CREEC is used to create an independent ECB for immediate or deferred processing. Core block of the specified data level and optionally the 104-byte variable sized parameters are passed to the created ECB.

Syntax:

Assembler:

CREEC Program,Dn,R/D

Program	Name of the program that is to be activated with the created ECB.
Dn	Specifies the data level (D0-DF) of the block to be passed to the created ECB.
R/D	ECB is placed in ready/deferred list.

C:

void creec(int length, const void *parm, void (*segname)(),

enum t_lvl level, int priority);

length	An integer containing the number of bytes to be passed to the created ECB. A value of zero indicates that no parameters will be passed. A maximum of 104 bytes may be passed.
parm	A pointer to type *void* to the parameters to be passed.
segname	For *creec*, a pointer to the external function to be called. *segname* must map to an assembler segment name, a TPF segment name or a TPF ISO-C dynamic load module (DLM) name.
level	One of 16 possible values representing a valid data level from the enumeration type t_lvl, expressed as Dx, where x represents the hexadecimal number of the level (0–F). The working storage block on this data level is the block to be detached.
priority	Use defined terms *CREEC_DEFERRED* to indicate placement on the deferred list and *CREEC_IMMEDIATE* to indicate placement on the ready list.

Other variants in C:

void	_CREEC(int length, const void *parm, const char *segname, enum t_lvl level, int priority);

void	creec(int length, const void *parm, void (*segname)(), TPF_DECB *decb, int priority);
void	_CREEC(int length, const void *parm, const char *segname, TPF_DECB *decb, int priority);

Entry Conditions:

- There must be a core block present on the specified data level, if no level is specified then D0 should contain a core block.
- R14 must contain the number of bytes of parameters to be passed to the created ECB work area.
- R15 must contain the address of the start of the parameters which are to be passed.

Return Conditions:

Assembler:

- Control is returned to NSI.
- Core block is unavailable for the use by the ECB that issued the CREEC macro. The specified CBRW is set to indicate the absence of core block.
- Register R14, R15 contents are unknown and other registers are preserved across this macro call.

Exception:

CTL- 14	Length of the parameter list is more than the maximum length allowed.
CTL- D0	Level specified is not holding a block.

Programming Consideration:

- The core block in the specified level is released when the macro is processed. Once the core block is released, references to it cause errors.
- The specified program receives control.

Example:

Assembler:

```
LA R14,0
CREEC XYZ4,D4,R
```
Creates entry to program XYZ4 with Contents of D4 placed in D0 of the new ECB.

C:

```
void ASM0();
char *parmstring=”101/15AUG”;
creec(strlen(parmstring),parmstring,ASM0,
      D0,CREEC_IMMEDIATE);
```
The above example creates an ECB dispatched from the ready list for program ASM0, passing string 101/15AUG and the core on level D0 as input data to the program.

CRETC – Create a Time-Initiated Entry

CRETC is used to create an entry to a program after a specified interval of time has elapsed. CP will save this request and transfer control to the specified program at the correct time with a 4-byte parameter and an optional data level as specified.

Syntax:

Assembler:

CRETC M/S,Program,LEVEL=Dn

M/S	Indicates the specified time increment is in minutes or seconds.
Program	Name of the program that is to be activated with the created ECB.
LEVEL	Data level (D0-DF) of the block to be passed to the created ECB.

C:

void cretc(int flags, void (*segname)(), int units, const void *action);

Flags	Logical or (I) of the following bit flags that are defined in tpfapi.h: ***CRETC_SECONDS or CRETC_MINUTES*** indicates whether the *units* parameter is expressed in seconds or minutes. Code only one of these options. ***CRETC_1052*** is an optional parameter that permits the program specified by *segname* to be activated at the time requested, even when the system is in 1052 state. If CRETC_1052 is not specified, the program cannot be activated until the system is cycled above 1052 state.
segname	For *cretc*, a pointer to the external function being called. *segname* must map to an assembler segment name, a TPF segment name or a TPF ISO-C DLM name.
Unit	An integer value defining the time increment in minutes or seconds that will elapse before activating the ECB.
Action	Treated as a pointer to type *void,* the first 4 bytes of which are passed to the created ECB as an action word in EBW000-003.

Other variants in C:

void	_CRETC(int flags, const char *segname, int units, const void *action);

Entry Conditions:

- R14 must contain the time increment value in the rightmost 3 bytes.
- R15 must contain a 4-byte action word that will be passed to the called program.
- There must be a core block on the ECB data level specified.

Return Conditions:

Assembler:

- Control is returned to NSI.
- Core block is unavailable for the use by the ECB that issued the CRETC macro. The specified CBRW is set to indicate the absence of core block.
- Register R14, R15 contents are unknown and other registers are preserved across this macro call.

Exception:

CTL-E	CRET table full.
CTL-D0	Level specified is not holding a block.

Programming Consideration: None

Example:

Assembler:

```
XR   R14,R14
ICM R14,B’0111’,=XL3’60’
L    R15,=C’ALRM’
CRETC S,XYZ5,LEVEL=D7
```

Creates entry to program XYZ5 after 60 seconds with contents of D7 placed in D0 of the new ECB.

C:

```
void ASM0();
cretc(CRETC_SECONDS,ASM0,5,”INIT”);
```

The above example creates a new ECB for program ASM0 after 5 seconds have elapsed. the 4-byte character string INIT is placed in EBW000-003 of the new ECB.

CPROC – Define a C function prototype for the CALLC

Define a prototype for a C function called by assembler program.

Syntax:

Assembler:

CPROC RETURN= rtype, fname,(parmTypes)

RETURN	Specifies the data returned for a C function, where rtype is a return type. (p pointer, f floating point)
fname	C Function name.
parmTypes	Is the parameter type for C function.

Entry Conditions:

- You must specify the CPROC macro to define the prototype for the C function before using the CALLC, ENTDC, ENTNC, or ENTRC macro to call a C function. Add the CPROC statement directly to the beginning of the assembler program; it applies to all of the matching CALLC, ENTDC, ENTNC, and ENTRC statements that follow. Alternatively, you can add the CPROC statement to the UPROC macro to avoid updating the assembler program.
- Floating-point data must be in binary format if specified with the CALLC macro.

Return Conditions:

None

Programming Consideration:

- A structure is not supported as a return type. Use a pointer to return the address of the structure.
- For parameters, use a pointer to pass the address of a structure.

Example:

Assembler:

```
   CPROC  RETURN=p,  buildRecord,(i,i,i)  where  p  =  pointer,
buildRecord = fname and 3 integer parameters.
```

CALLC – Call a C function

Use this general macro to call a C function from a basic assembler language (BAL) program.

Syntax:

Assembler:

CALLC fname,label1,freg,reg,regaddr,AMODE

Fname	is the C function name.
Label1	is the label of an area of storage that contains a parameter.
Freg	is a floating-point register (F0-F15) that contains data in binary floating-point format.
Reg	is a register that contains a parameter.
Regaddr	is a register that contains the address of the parameter.
AMODE	specifies the format that is used by all address parameters. The valid address format values are: ***amodedef*** The default value is specified by the AMODEDEF parameter of the BEGIN macro. If AMODEDEF is not specified on the BEGIN macro, the default value is 31. **31** indicates that addresses are specified in 31-bit format. **64** indicates that addresses are specified in 64-bit format.

Other variants in C: None

Entry Conditions:

- R9 must point to the entry control block (ECB) address.
- R15 cannot be used as an input parameter register.
- You must define the prototype for the C function by using the CPROC macro before you use the CALLC macro.
- Floating-point data passed as a parameter must be in binary-floating-point format.

Return Conditions:

Assembler:

- If the C function returns floating-point data, F0 contains the return value.
- If the C function returns void or floating-point data, the contents of R15 are unknown.
- If the C function does not return floating-point data, R15 contains the return value.

Exception: None

Programming Consideration:

- The number of parameters specified using the CALLC macro must be equal to the number of parameters specified with the corresponding CPROC macro.
- A structure is not supported as a return type. Use a pointer to return the address of the structure.
- For parameters, use a pointer to pass the address of a structure. Alternatively, you can pass a structure itself as a parameter by using a label of storage that contains the parameter. Do not pass the structure itself inside a register. See the CPROC macro for examples of passing parameters.

Example:

Assembler:

The following example shows an assembler program calling a C function that uses three parameters. The CPROC and CALLC macros are issued to call the C function.

```
      CPROC RETURN=i,myfname,(c,p,us)

...
      GETCC D1,L1       Get a core block
      LR    R2,R14      Copy address of core block
      LA    R3,=H'40'   Get address of 40 in R3
      CALLC myfname(parm_char,R2,(R3))
...
parm_char   DC    C'C'  Character passed to C function
      END
```

Q1. What are the macros can be used to put ECB in ready list?

Q2. What are the macros can be used to put ECB in defer list?

Q3. Program ABCD has 4 functions those are Entry Validation, Data verification, Database update and response in sequence. Due to so many enhancements size of ABCD is almost 4K so we need to split the ABCD. For below scenarios which macro is advisable to use to call EFGH segment.

1. Keep Entry Validation and Data verification in ABCD and database update and response in EFGH segment then stops the processing by EXITC.

2. Code Entry validation and data verification in EFGH whereas database update and response in ABCD segment then continues process by BACKC.

3. Keep Entry Validation and Data verification in ABCD and database update and response in EFGH segment then continues the processing by BACKC.

7. Inter-process Communication

7.1 Introduction

- In zTPF, the internal Event facility is used to achieve synchronization between the ECBs.

- Such synchronization is utilized in applications where several ECBS are created to work on individual sub-processes while the master ECB that triggers the ECBs tracks the overall progress of the process.

- Typically, the master ECB creates a child ECB to complete a subtask, and waits on a unique Event while the child ECB completes the task and posts back to the master. The master then resumes execution.

7.2 Internal Event Facility

- The Event facility allows the application programs to define a unique event, which can be waited on and posted by other ECBs.

7.3 Defining an Event

- Events are defined using the EVNTC macro.

- An event is uniquely identified by an 8-character name which can be either application or system defined.

- The EVNTC macro causes the control program to register the event in the internal Event Table.

- The parameters for an EVNTC macro is usually supplied in the CBRW and FARW fields of a data level specified in the LEVEL parameter.

7.3.1 Event Name

- An event name can be either decided by the application or the application can let the system to generate one.
- This is controlled by the NAME parameter of the EVNTC macro.
- Application defined event names must be supplied in the CE1FAx field of the data level specified.
- When the specified event name already exists in the event table, the control branches to the label specified in the DUPNAM parameter of the macro.

7.3.2 Event Type

- The Event facility allows four types of trigger mechanism, of which the most commonly used, is the CNT, known as count type mechanism.
- It allows the application to specify the count of posts required in order to complete the event.
- The count (usually 1) is supplied in the first two bytes of the field CE1CRx of the data level specified.

7.3.3 Timeout

- The application can specify a timeout limit of up to 32,768 seconds using the TIMEOUT parameter.
- If the event is not posted within the specified limit, the event is considered to be in error.

7.3.4 Waiting for an event

- After defining the event, an ECB can wait on the event by issuing the EVNWC macro.

- The name of the event must be supplied in the CE1FAx field of the data level specified.

- If the event is not found in the internal Event Table, the control program causes the control to branch to the label specified in the NFOUND parameter. If not, the ECB loses control and waits for the event to be posted (or timed- out).

7.3.5 Posting on an event

- The events are posted back using a POSTC macro.

- The name of the event must be supplied in the CE1FAx field of the data level specified.

- If the event is not found in the internal Event Table, a non-zero condition code is set and the control branches to the label specified in the NFOUND parameter.

- If not, the count in the event is decremented by one. When the count reaches zero, the ECB that waits on the event is resumed.

7.3.6 Delaying an ECB

- The Event facility can be used to delay execution of an ECB by a specified number of seconds.

- This is achieved by using the TIMEOUT parameter in the EVNTC macro.

- When the EVNWC macro is processed, the ECB will be delayed by the amount of time specified. POSTC macro is not required to be coded in this case.

7.4 Inter-process communication macros

EVNTC – Define Internal Event

EVNTC is used to define an event.

Syntax:

Assembler:

EVNTC LEVEL=Dn,TYPE=CNT,NAME=Y/N,DUPNAM=label,TIMEOUT=N/Rx

LEVEL	Specifies the data level (D0-DF).
TYPE	**CNT** – Counter type event
NAME	**Y** – Indicates that the event name is supplied by the application in the FARW of the data level specified.
	N – Indicates the macro processor to generate a unique event name and return it in the FARW of the data level specified. Default is N if not coded.
DUPNAME	A label to branch to if the specified name already exists in the event table.
TIMEOUT	**N** – Self-defining value ranging in between 0 to 4096.
	Rx – Register (R0-R7, R14, R15) ranging from 0 to 32,768.
	If a value of 0 is specified, time-out is not performed. Defaults to 180 seconds if not coded.

C:

```
int evntc(struct ev0bk      *evninf,
          enum t_evn_typ evtyp,
          char                evn_name,
          int                 timeout,
         enum t_state       evstat);
```

evninf	A pointer to the *evntc parameter block.*
evtype	The type of event being defined. The argument must belong to the enumerated

	type t_evn_typ defined in tpfapi.h. Use one of the predefined terms: **EVENT_MSK** for mast events. **EVENT_CNT** for count events. **EVENT_CB_Dx** where x is a single hexadecimal digit (0-F) for core events. **EVENT_LIST** for list events. A counter event is complete when the specified count becomes zero. The *postc* Function decreases the event count by 1. A mask event is complete when the mask is completely reset. The postc function uses a 16-bit mask to reset the mask bits. Core block events are completed after the first *postc* function call. A list event is completed when all the data items have been posted.
evn_name	Indicates if the caller has supplied the event name. If Y is specified, the event name is supplied by the caller in evninf->evnbkn. If N is specified, a unique event name is generated by the function processor and returned in evninf->evnbkn.
timeout	An integer specifying the number of seconds an ECB waits before the event is assumed to be in error. The timeout parameter is a value with a range of 0 to 65535. If 0 is specified as the time-out, the event will not time out.
evstate	Indicates whether the event is run in NORM state only or in all states. Code EVNTC_1052 if the event can run in all states or EVNTC_NORM if the event can only run in NORM state.

Other variants in C: None.

Entry Conditions:

- Event name must be specified in CE1Fax field of the data level specified unless NAME=N is coded.
- The count value for the event must be specified in the first two bytes of CR1CRx field.

Return Conditions:

Assembler:

- Control is returned to the NSI unless the specified name is not unique, in which case the label for DUPNAM is branched to.
- The specified level is available for reuse and the CBRW contents remain unchanged. FARW contains the name of the event.
- Register R14, R15 contents are unknown and other registers are preserved across this

macro call.

C:

- An integer value of 0 or 1 is returned where 0 is normal return & 1 is event name already exists.

Exception:

CTL- E00	The data level specified is in use.

Programming Consideration: None

Example:

Assembler:

```
MVC CE1FA4,=CL8'EVENT001'
MVC CR1CR4(2),=X'1'
EVNTC LEVEL=D4,TYPE=CNT,NAME=Y,DUPNAM=ERR_RTE,
      TIMEOUT=5
```

Count type event with a timeout limit of 5 seconds.

C:

```
Struct ev0bk event_blk;
event_blk.evnpstinf.evnbkc1=1;
evntc(&event_blk,EVENT_CNT,'N',250,EVNTV_1052);
```

Defines a count event to be named by the system with a time-out of 250 seconds and capable of running in all system states.

EVNWC – Wait for Event Completion

EVNWC is used to wait for the completion of an event defined by an earlier EVNTC macro.

Syntax:

Assembler:

EVNWC LEVEL=Dn,TYPE=CNT,ERROR=label1,NFOUD=label2

LEVEL	Specifies the data level (D0-DF).
TYPE	**CNT** – Counter type event.
ERROR	A label to branch to if the event has completed with error.
NFOUND	A label to branch to if the event does not exist in event table.

C:

int evnwc(struct ev0bk *evninf,enum t_evn_typ type);

evninf	A pointer to the evnwc parameter block.
type	The type of event being completed. The argument must belong to the enumerated type t_evn_typ defined in tpfapi.h. Use one of the predefined terms: **EVENT_MSK** for mast events. **EVENT_CNT** for count events. **EVENT_CB_Dx** where x is a single hexadecimal digit (0-F) for core events. **EVENT_LIST** for list events.

Other variants in C: None**.**

Entry Conditions:

- Event name must be specified in CE1FAx field of the data level specified.

Return Conditions:

Assembler:

- Upon normal event completion, control is returned to NSI.

- If the event completed with error, control is returned to the label specified by the ERROR parameter.
- If the specified event is not found an immediate return is made to the label specified by the NFOUND parameter.
- The CBRW of the data level specified is modified but FARW remains unchanged.
- Register R14, R15 contents are unknown and other registers are preserved across this macro call.

C:

- An integer value of 0, 1 or 2 is returned where 0 is normal return, 1 is event name already exist and 2 is event is an error post.

Exception:

CTL- E03	Event Type Mismatch.

Programming Consideration:

- The EVNWC causes unconditional loss of control of the ECB unless the named event is not found.
- The EVNTC and EVNWC macros can be used to introduce a delay as explained in the previous section.

Example:

Assembler:

Wait on the event named EVENT001.

```
MVC CE1FA4,=CL8’EVENT001’

EVNWC LEVEL=D4,TYPE=CNT,ERROR=EVNTERR,NFOUND=EVNTNFD
```

C:

Defines an event and waits for the event to complete.

```
Struct ev0bk event_blk;
evnwc(&event_blk,EVENT_CNT);
```

POSTC – Mark Event completion

POSTC macro is used post completion of an event.

Syntax:

Assembler:

POSTC LEVEL=Dn, TYPE=CNT, NFOUND=label1

LEVEL	Data level (D0-DF).
TYPE	CNT – Counter type event.
NFOUND	A label to branch to if the event is not outstanding.

C:

int postc(struct ev0bk *evninf,enum t_evn_typ type,int ercode);

evninf A pointer to the *postc* parameter block.

Type The type of event element being completed. The argument must belong to the enumerated type *t_evn_typ*, defined in *tpfapi.h*. Use the following predefined terms:

EVENT_MSK

for mask events

EVENT_CNT

for count events

EVENT_CB_D*x*

where *x* is a single hexadecimal digit (**0**–**F**) for core events

EVENT_LIST

for list events.

ercode Indicates if the post is an error post and causes the event completion to be signaled as completed with error. A value of zero indicates that the post was not an error post. A nonzero value contains the error code. The error code can range from 1–256.

Other variants in C: None

Entry Conditions:

- Event name must be specified in CE1FAx field of the data level specified.

Return Conditions:

Assembler:

- Control is returned to the NSI and CC will be set as follows:

  Zero – event found and processed.

  Non – Zero, event not found.
- The current count value of the event is decremented by one.
- R14 and R15 are unknown; other registers are preserved across this macro call.

C:

- Integer value of zero. If **EVENT_CB_D*x*** or **EVENT_LIST** is coded for *type*, the specified core level is marked as not holding a block.
- An integer value of 1 is returned if the event name does not exist.
- An integer value of 2 is returned if a list event data item is not found.

Exception:

Ctl-E03	Event Type Mismatch

Programming Consideration: None

Example:

Assembler:

```
MVC CE1FA4,=CL8'EVENT001'
```
Post on the event named EVENT001.

```
POSTC LEVEL=D4,TYPE=CNT,NFOUND=EVNTNFD
```

C:

```
struct ev0bk pb = { { "my_event" } };
```
Post on the event named my event
```
postc(&pb, EVENT_CNT, 0);
```

Q1. What is the difference between Event and Create ECB macros?

8. Globals

8.1 Introduction

- Globals (not to be confused with global variables in C) provides an efficient mechanism to access frequently accessed units of data.

- The Globals are maintained in the main memory and hence aid faster retrievals.

- It’s typical usage includes:
  - Storing frequently accessed data such as system date.
  - Sharing data between ECBs
  - Maintaining application activation switches

- To preserve the integrity of data in Globals, they are also maintained in the file storage.

- The data is kept in sync by a mechanism called key-pointing. The Globals are initialized and loaded from the file storage at the system restart.

- The Globals are maintained in fixed locations in the main memory, identified as Global areas 1, 2, and 3. Each of these areas contains several Global records which is nothing but a logical collection of data with its own attributes.

- The Global records are further subdivided into fields of size ranging from 1 to 256 bytes, referred to as Global Fields.

- Apart from the Global areas 1, 2 and 3, there is also an Extended Global area that reside above the 16-MB boundary, available only for applications that runs on 31-bit addressing mode.

8.2 Access

- The Globals fields of area 1 and 3 are defined in the GLOBA and GLOBY macro respectively.

- The GLOBZ macro is used to establish the addressability to the macro. After issuing the GLOBZ macro, the Global field names can be used to access the fields.

8.3 Update

- Global fields reside in an area of main storage that carries a storage protection key different from that in the PSW used for application programs.

- Therefore, to update a Global field, the storage protection key in the PSW must be modified to match that of Global area containing the field.

- This is done by issuing the GLMOD macro after which the contents of the Global field can be modified.

- Once the update is complete, the protection key in the PSW must be restored by issuing a KEYRC macro.

- If the Global is key-pointable, then a FILKW macro must be issued to key-point it and restore the protection key in the PSW.

8.4 Global macros

GLOBZ – Define Global fields

GLOBZ calls the Global definition macros for Global areas 1 and 3 and establishes addressability to a Global area in a register specified.

Syntax:

Assembler:

GLOBZ FLD=symbol, REGR=Ry, REGS=Rz

FLD	If specified, the symbol is used determine the Global area in which it is stored.
REGR	If specified, the register Ry is loaded with the address of GLOBA.
REGS	If specified, the register Rz is loaded with the address of GLOBY.

C:

void *glob(unsigned int tagname);

Tagname	This argument, which must be defined in *c_globz.h*, is treated as an unsigned 32-bit integer that describes the displacement, length, and attributes of the global field or record.

Other variants in C: None

Entry Conditions:

- At least one REGx parameter must be coded.

Return Conditions:

Assembler:

- Control is returned to the next sequential instruction.
- The registers defined as the operands of the REGx keywords are loaded with the addresses of the appropriate global areas.
- The contents of all other registers are preserved across this macro call.

C:

- A global address. If the tagname parameter describes a global field, the address of the field is returned. If the tagname parameter describes a global record, the address of the global directory for that record is returned. A global directory address can point to either the main storage address or the file address of the record, depending on how globals are defined in your system. If the tagname describes the main storage address directory for a global record, an additional level of indirection is required to address the global record itself.

Exception: None

Programming Consideration:

- When the FLD parameter is specified, the address of the Global area where the field resides is loaded into the register specified regardless of the register keyword supplied.

Example:

Assembler:

```
GLOBZ FLD=@UNICNT,REGR=R15
LH   R14,@UNICNT
```

Gets addressability to Global area where @UNICNT is defined in register R15.

Loads the two byte content in @UNICNT to register R14.

C:

```
long int *longglob_ptr = glob(_lfld);
```

glob function returns the address of a 4 byte global field (which will be treated as a long int).

GLMOD – Change global protect key

GLMOD macro changes the storage protection key in the PSW to match that of the Global area that is to be modified.

Syntax:

Assembler:

GLMOD

C:

int glob_modify(unsigned int tagname, void *dstptr,const void *srcptr, int length);

tagname	This argument, which is defined in header file *c_globz.h*, uniquely identifies the z/TPF global field or record to be accessed.
Dstptr	The address of the data in the global field or record to be modified.
Srcptr	the address of the data used to modify the global field or record.
Length	The length of the modified data.

Other variants in C: None

Entry Conditions: None

Return Conditions:

Assembler:

- Control is returned to the next sequential instruction.
- The protection key in the PSW is changed.
- R14 and R15 are unknown but the other registers are preserved across this macro call.

C:

- **GLOB_RC_OK-**The global field or record was updated successfully.

Exception: None

Programming Consideration:

- The protection key in the PSW must be restored after Global modification.

Example:

Assembler:

```
GLOBZ REGR=R15          BASE GLOBAL AREA 1
LH    R14,@UNICNT       RETRIEVE THE COUNTER
AHI   R14,1             INCREMENT IT
GLMOD                   GET GLOBAL UPDATE KEY
STH   R14,@UNICNT       UPDATE GLOBAL
....                    RESTORE PROTECTION KEY
```

C:

The following example modifies a synchronizable global record.

```
  #include<tpfglbl.h>
  #include<c$globz.h>
  ...
 {
      /**************************************************************/
      /*  Increment data element mysdata in synchronizable global record _mysglob.  */
      /**************************************************************/
   struct mysglbrec **msgrptrptr = glob_lock(_mysglob);
   struct mysglbrec *msgrptr = *msgrptrptr;
   long newdata = msgrptr->mysdata + 1;
   glob_modify(mysglob,&msgrptr->mysdata,&newdata,
            sizeof(msgrptr->mysdata);
   glob_sync(_mysglob);
 }
```

KEYRC – Restore protection key

KEYRC macro restores the storage protection key in PSW to match that of working storage.

Syntax:

Assembler:

KEYRC OKEY=YES

OKEY	Restores the original protection key that was saved by the previous CINFC macro call with the W option specified. The OKEY=YES parameter requires restricted authorization (OPTIONS=(RESTRICT)) on the IBMPAL macro.

C:

void keyrc(void);

Other variants in C: None

Entry Conditions: None

Return Conditions:

Assembler:

- Control is returned to NSI.
- The protection key in PSW is restored to that of working storage (ECBs, data blocks, and fixed unprotected core).
- R14 and R15 are unknown but other registers are preserved across this macro call.

C:

void.

Exception: None

Programming Consideration:

- Use immediately after altering a Global field.

Example:

Assembler:

```
GLOBZ REGR=R15          BASE GLOBAL AREA 1

LH    R14,@UNICNT       RETRIEVE THE COUNTER

AHI   R14,1             INCREMENT IT

GLMOD                   GET GLOBAL UPDATE KEY

STH   R14,@UNICNT       UPDATE GLOBAL
KEYRC                   RESTORE PROTECTION KEY
```

C:

The following code restores the storage protection key after a call to the cinfc function in which the CINFC_WRITE option was specified.

```
 char *field_ptr;
field_ptr = cinfc(CINFC_WRITE, CINFC_CMMINC);
keyrc();
```

FILKW – File Keyword

FILKW enables filing of global key-pointable fields or records. Optionally it restores the storage protection key in the PSW.

Syntax:

Assembler:

FILKW R/N,KeypointN,FLD=NO or YES

R/N	R – Restore protection key in the PSW.
	N – Don't restore.
Keypoint N	Names of the key-pointable Globals to be filed. N ranges from 1 to 8.
FLD	**YES** – Indicates one of the key-pointed records is a Global field.
	NO – Indicates only Global records are key-pointed. Default is NO.

C:

void tpf_filkw(enum t_filkw options,unsinged int number,unsigned int tagname,…);

options	Specifies the options associated with the keypoint request. Use one or more of the following defined terms. When specifying more than one term, use a + to separate the terms. **FILKW_RESTORE** restores storage protection. **FILKW_NO_RESTORE** does not restore storage protection. **FILKW_FIELD** one or more of the *tagname* parameters are filed names. You can specify either FILKW_RESTORE or FILKW_NO_RESTORE, but not both.
number	Specifies the number of records or field names that fellow. You can specify from 1-8.
tagname	Specifies the z/TPF global field, record or keypoint record to be pointed. *tagname…* specifies that a maximum of eight *tagname arguments can be*

	specified.

Other variants in C: None.

Entry Conditions:

- Global fields must have been defined.

Return Conditions:

Assembler:

- Control is returned to NSI.
- The protection key in PSW is restored to that of working storage if 'R' parameter is specified.
- If key-pointing is active the file copies will be updated.
- Register R14, R15 contents are unknown and other registers are preserved across this macro call.

Exception: None

Programming Consideration:

- Issue only on key-pointable Globals.

Example:

Assembler:

```
GLOBZ  REGR=R15              BASE GLOBAL AREA
LH     R14,@DCFLD1           RETRIEVE THE GLOBAL DCFLD1
AHI    R14,1                 INCREMENT IT
GLMOD                        GET GLOBAL UPDATE KEY
STH    R14,@DCFLD1           UPDATE GLOBAL
FILKW  R,@DCFLD1,FLD=YES     KEYPOINT GLOBAL AND
                             RESTORE PROTECTION KEY
```

C:

The following example requests keypointing of keypointable global field _ns1ns.

```
int *ns1ns;
ns1ns=(int*)glob(_ns1ns);
tpf_filkw((enum t_filkw) (FILKW_RESTORE+FILKW_FIELD),
           _ns1ns);
```

GFGATC – Obtain format-2 global attribute table entry

Use this system macro to obtain addressability for the attributes of format-2 global records.

Syntax:

Assembler:

GFGATC NAME=GLOBAL,REG=reg,ERROR=LABEL,AMODE=AMODEDEF

NAME	specifies the name, where: ***`global`*** is the 1- to 8-character name of the format-2 global record. ***`reg`*** is a general register (E-type: R0–R8; C-type: R1–R12) that contains the address of the name of the 8-character format-2 global record (padded on the right with blanks).
REG	specifies the general register (E-type: R0–R8, R14, or R15; C-type: R1–R12, R14, or R15) that is to contain the address of the requested format-2 global attribute table (FGAT) entry on return.
ERROR	specifies an error routine, where *label* is the location where control is passed if the name of the requested format-2 global record is not valid.
AMODE	specifies the addressing mode of the calling application. This parameter is not allowed when called from the control program (CP). The following values are allowed: **`amodedef`** is the default value that is specified by the AMODE parameter on the BEGIN macro. If AMODE is not specified on the BEGIN macro, the default value is 31. **`31`** indicates that the caller is running in 31-bit addressing mode. **`64`** indicates that he caller is running in 64-bit addressing mode.

C:

FGAT_ENTRY *tpf_gfgatc (char *globalname);

Globalname	The 8-byte name of the requested format-2 global record.

Other variants in C: None

Entry Conditions:

- For E-type programs, R9 must contain the address of the entry control block (ECB) that is being processed.
- For C-type programs, R0 contains the subsystem user (SSU) ID in bytes 6–7.

Return Conditions:

Assembler:

- The register specified on the REG parameter contains the address of the requested FGAT entry, or zero if the specified name of the format-2 global record is not valid.
- The contents of R14 and R15 are not preserved across this macro call.

C:

- The address of the requested FGAT entry.
- Error return: NULL

Exception: None

Programming Consideration:

- This macro is intended for use by system programs only. Application programs have no need for the information contained in the FGAT entry associated with each format-2 global record.
- Application programs normally use the GLOBLC macro with the FUNC=STAT parameter specified to request information about a format-2 global record.
- This macro can be used only with z/TPF format-2 globals. To request addressability to the attributes of z/TPF format-1 globals, use the IGATC macro.

Example:

Assembler:

The following example obtains the address of the FGAT entry for the _TPFAUX1 format-2 global record and returns the address in general register R6.

```
GFGATC NAME=_TPFAUX1,REG=R6,ERROR=ERROR_LABEL1
```

C:

The following example obtains the address of the FGAT entry for the _TPFAUX1 format-2 global record.

```
#include <tpf/tpfglbl.h>

FGAT_ENTRY *fgatPtr;
fgatPtr = tpf_gfgatc ("_TPFAUX1");
```

GLBLUC – Manage format-2 global records

Use this system macro to change the current location in main storage of the specified format-2 global record.

Syntax:

Assembler:

GLBLUC NAME=GLOBAL,PARMS=REG,RC=REG2,FORCE=NO/YES,AMODE=AMODEDEF,

BACKUP=YES/NO,SSUGROUP=YES/O

NAME	specifies the name, where: ***`global`*** is the 1- to 8-character name of the format-2 global record. ***`reg`*** is a general register (E-type: R0–R8; C-type: R1–R12) that contains the address of the name of the 8-character format-2 global record (padded on the right with blanks).
PARMS	specifies the location of the input parameter list, where *reg* is a general register (R0–R8) that contains the address of the input parameter list.
RC	specifies the register, where *reg2* is the general register (R0–R8, R14, or R15) that will contain the return code on return.
FORCE	specifies whether the update request will be processed if the specified format-2 global record is initialized already, where: **`NO`** indicates that the update will not proceed if the global is initialized already. **`YES`** indicates that the update will proceed even if the global is initialized already.
AMODE	specifies the addressing mode of the calling application, where the following values are allowed: **`amodedef`** is the default value that is specified by the AMODE parameter on the BEGIN macro. If AMODE is not specified on the BEGIN macro, the default value is 31. **`31`** indicates that the caller is running in 31-bit addressing mode. **`64`** indicates that the caller is running in 64-bit addressing mode.
BACKUP	specifies whether the previous main storage address will be saved if this global is initialized already. **`YES`** indicates that the previous main storage address will be saved

	in a backup location pointed to from within the format-2 global attribute table (FGAT) entry of the global record. `NO` indicates the previous main storage address will not be saved.

SSUGROUP	specifies whether the requested update will be applied to all members of an SSU group if the subsystem user (SSU) where the ECB is processing is a member of an SSU group that this global record includes.
	`YES`
	indicates that the update will be made to all members of the SSU group.
	`NO`
	indicates that the update will only be made to the SSU where the ECB is processing, even if this SSU is a member of an SSU group that is included by this global record.

C:

int tpf_glbluc (char *globalname, TPF_GLBLUC_PARMS *parms,enum t_glforce force);

Globalname	The 8-byte name of the requested format-2 global record.

PARMS	A pointer to a parameter block described by the **TPF_GLBLUC_PARMS** structure. All input fields in this block must be initialized before this function is called.
	`is`
	The I-stream that this request is being made for. To indicate all I-streams, a value of 0 must be specified. This parameter only has meaning when the format-2 global record specified is defined as I-stream unique.
	`address`
	The new main storage address of the specified format-2 global record.
	`size`
	The new size of the specified format-2 global record.
	`user_data`
	As many as 72 bytes of user data for the specified format-2 global record.

FORCE	Specifies whether the update request will be processed if the specified format-2 global record is already initialized, where: **`GLFORCE_NO`** Indicates that the update will not proceed if the global is initialized already. **`GLFORCE_YES`** Indicates that the update will proceed even if the global is initialized already. **`GLFORCE_NO_BACKUP`** Indicates that the previous main storage address will not be saved if this global is already initialized. This option is specified in conjunction with either the GLFORCE_NO or GLFORCE_YES option, separated by a + (plus sign). If this option is not specified, the previous main storage address will be saved in a backup location, pointed to from within the FGAT entry of the global. **`GLFORCE_NO_SSUGROUP`** Indicates that the update should be made only to the subsystem user (SSU) where the ECB is processing, even if the SSU where the ECB is processing is a member of an SSU group that is included by this global record. This option is specified in conjunction with one or more of the other GLFORCE_ options, separated by a + (plus sign). If this option is not specified, and the SSU where the ECB is processing is a member of an SSU group included by this global record, the update will be applied to all members of that SSU group.

Other variants in C: None

Entry Conditions:

- R9 ***must*** contain the address of the ECB being processed.
- The subsystem user (SSU) where the ECB is processing will be used to determine the copy of the storage that is to be updated for this global record. This is particularly relevant when the global record is subsystem-user unique.
- The input parameter list, as specified by the PARMS parameter, ***must*** be in the following format, mapped by DSECT IGLBLU:

  IGLBLU_ADDR

  The new location in main storage of the format-2 global record.

  IGLBLU_SIZE

  The new size of the format-2 global record.

IGLBLU_IS

The I-stream that this request is being made for. To indicate all I-streams, a value of 0 must be specified. This parameter only has meaning when the format-2 global record specified is defined as I-stream unique.

IGLBLU_USER

As many as 72 bytes of user data for the specified format-2 global record.

Return Conditions:

Assembler:

- Control is returned to the next sequential instruction (NSI).
- The register specified on the RC parameter contains one of the following values:

  GLBLUC_OK

  The requested function completed successfully.

  GLBLUC_NAME_NOT_VALID

  The specified global does not specify the name of a valid format-2 global record.

  GLBLUC_INITIALIZED

  The specified global is already initialized, and the FORCE=NO parameter was specified. No update was made.

  GLBLUC_INTERNAL_ERROR

  An internal error occurred while attempting to update the global. No update was made.

  GLBLUC_IS_NOT_VALID

  The I-stream specified is not valid. No update was made.

C:

- GLBLUC_OK.
- **`GLBLUC_FORCE_NOT_VALID`**
  - The option specified on the **force** parameter is not valid.
- **`GLBLUC_NAME_NOT_VALID`**
  - The specified global does not specify the name of a valid format-2 global record.
- **`GLBLUC_INITIALIZED`**
  - The specified format-2 global record is already initialized and **GLFORCE_NO** was specified.
- **`GLBLUC_INTERNAL_ERROR`**
  - An internal error occurred while attempting to update the global.
- **`GLBLUC_IS_NOT_VALID`**
  - The I-stream specified is not valid.

Exception:

06410A	UNAUTHORIZED USE OF FORMAT-2 GLOBAL UPDATE

Programming Consideration:

- This macro can be run on any I-stream.
- This macro enables you to replace the complete data for a format-2 global record with updated data without disrupting the online system.
- This macro is intended to allow system programs to update the location in main storage of a specified user-controlled format-2 global record. Alternatively, the ZGLBL GLOBAL INITIALIZE command is provided to update format-2 global records.
- The specified format-2 global record must first be opened for reading and writing with the GLOBLC macro with the FUNC=OPEN parameter specified before calling this routine. Otherwise, results cannot be predicted.
- When GLBLUC macro processing has completed successfully, the FGAT entry for the specified format-2 global record has been updated to indicate the new location of the record in main storage, and the user data supplied has been saved in the user data section. The previous information is moved to a backup location in the FGAT entry so that it can still be referenced by user applications. When it is determined that no ECBs in the system are still referencing the previous data address, storage needs to be released. Storage is released automatically by the z/TPF system for system-controlled format-2 global records. It is your responsibility to provide a similar mechanism for releasing user-controlled format-2 global records.
- The user data section of the input parameter block will be stored in the corresponding user data section of the FGAT entry. You can use this field, for example, to indicate the location of the global record on DASD for user-controlled format-2 global records, such as a fixed file location, file system file, or z/TPF collection support (z/TPFCS) persistent identifier (PID).
- This macro ***cannot*** be issued from the control program.
- This macro can only be used with z/TPF format-2 globals.

Example:

Assembler:

The following example updates the FGAT entry for the _USRAUX1 format-2 global record with the new location of the record in storage and on DASD.

```
      USING IGLBLU, EBW000
      MVC   IGLBLU_IS(2),CE1ISN           INDICATE CURRENT I-STREAM
      STG   R5,IGLBLU_ADDR                SET NEW MAIN STORAGE ADDRESS
      STG   R6,IGLBLU_SIZE                SIZE OF FORMAT-2 GLOBAL RECORD
      XC    IGLBLU_USER(72),IGLBLU_USER   CLEAR USER DATA AREA
      GLBLUC NAME=_USRAUX1,PARMS=EBW000
```

C:

The following example updates the FGAT entry for the _USRAUX1 format-2 global record with the new location of the record in storage and on DASD.

```
#include <tpf/c_f2glob.h>
int rc;                          /* Function return code */
TPF_GLBLUC_PARMS *parms;           /* tpf_glbluc input parameters */
void *new_addr;                  /* New main storage location */
parms = &(ecbptr()->ebw000);
parms->is = ecbptr()->ce1isn;
parms->address = new_addr;
parms->size= 4000;
memset (parms->user_data, 0, sizeof (parms->user_data));
rc = tpf_glbluc ("_USRAUX1", parms, GLFORCE_NO);
if (rc != GLBLUC_OK)
{
      printf ("An error occurred in tpf_glbluc\n");
      exit (0);
}
```

GLOBLC – Manage format-2 global records

Use this general macro to perform the following actions:

- Open a format-2 global record
- Change the open characteristics of a format-2 global record
- Write updates to a format-2 global record
- Close a format-2 global record
- Request information about a format-2 global record.

Syntax:

Assembler:

GLOBLC FUNC=CLOSE/CNTL/OPEN/STAT/WRITE,OPT=UPDATE/UPDATEWT/NOUPDATE/

RDONLY/RDFAST/RDWRITE/UNLOCK/UNLOCKWT,NAME=GLOBAL,ADDR=REG,

OFFSET=REG,LENGTH=REG,DESC=REG,AMODE=AMODEDEF

FUNC	specifies the requested format-2 global function, where the following values are allowed: **CLOSE** closes a format-2 global record. **CNTL** changes how a format-2 global record was opened. **OPEN** opens a format-2 global record for reading or for reading and writing. **STAT** requests information about an open format-2 global record. **WRITE** requests an update to an open format-2 global record.
OPT	specifies the options to be used for the requested format-2 global function, where the following values are allowed: **UPDATE** writes an update to the persistent copy and closes the format-2 global record. This option is valid only when FUNC=CLOSE is specified. **UPDATEWT** writes an update to the persistent copy, waits for completion, and closes the format-2 global record. This option is valid only when FUNC=CLOSE is specified. **NOUPDATE** closes the format-2 global record without writing updates to the persistent copy. This option is valid only when FUNC=CLOSE is specified. **RDONLY** opens for reading only. This option is valid only when FUNC=OPEN is specified. **RDFAST**

	opens for reading using only the fast path. Use this option when fast access is required; however, the z/TPF system will not track the global as open for this ECB. This option is valid only when FUNC=OPEN is specified. **`RDWRITE`** opens for reading and writing. The global will be locked for exclusive use by this ECB. This option is valid only when FUNC=OPEN or FUNC=CNTL is specified. **`UNLOCK`** unlocks a global record that was opened previously for read and write access. If this global record can be synchronized and updates had been written previously to the persistent copy of the global, synchronization notification will be sent to the other processors in the z/TPF complex. This option is valid only when FUNC=CNTL is specified. **`UNLOCKWT`** unlocks a global record that was opened previously for read and write access. If this global record can be synchronized and updates had been written previously to the persistent copy of the global, synchronization notification will be sent to the other processors in the z/TPF complex and this ECB will not get back control until all processors acknowledge that the synchronization is completed. This option is valid only when FUNC=CNTL is specified.
NAME	specifies the name of the format-2 global record, where: ***`global`*** is the 1- to 8-character name of the format-2 global record to be acted on. ***`reg`*** is a register (R1–R8) containing the address of the 8-character name of the format-2 global record, padded on the right with blanks.

ADDR	indicates the register (R0–R8, R14) that is to contain the address of the requested format-2 global record on return, when FUNC=OPEN is specified. When FUNC=STAT is specified, this indicates the register (R0–R8, R14) that contains the address of the IGLST parameter list that is to contain the requested information on return. The IGLST parameter list contains the following information:
	`IGLST_ADDR`
	is the main storage address of the format-2 global record.
	`IGLST_SIZE`
	is the size, in bytes, of the format-2 global record.
	`IGLST_SSU`
	contains a nonzero value if the format-2 global record is SSU-unique.
	`IGLST_IS`
	contains a nonzero value if the format-2 global record is I-stream unique.
	`IGLST_PROC`

	contains a nonzero value if the format-2 global record is processor unique. **IGLST_KEYPT** contains a nonzero value if the format-2 global record can be keypointed. **IGLST_SYNC** contains a nonzero value if the format-2 global record can be synchronized. **IGLST_PROT** contains a nonzero value if the format-2 global record is in protected main storage. **IGLST_RDONLY** contains a nonzero value if the format-2 global record is open for read-only access.
OFFSET	specifies the starting offset into the global record where the global update will occur, where *reg* is a register (R0–R8). Data that precedes the specified offset might not be included in the update request when the designated global is filed.
LENGTH	specifies the number of bytes that are to be included in the update request, where *reg* is a register (R0–R8). This parameter can be used only when the OFFSET parameter is specified.
DESC	contains the global descriptor on return when FUNC=OPEN is specified, where *reg* is a register (R0–R8). Otherwise, it indicates the register (R0–R8) that contains the global descriptor, which identifies the global record to be acted on. This parameter is required unless FUNC=OPEN,OPT=RDFAST is specified.
AMODE	specifies the addressing mode of the calling application, where the following values are allowed: **amodedef** is the default value that is specified by the AMODE parameter on the BEGIN macro. If AMODE is not specified on the BEGIN macro, the default value is 31. **31** indicates that the caller is running in 31-bit addressing mode. **64** indicates that the caller is running in 64-bit addressing mode.

C:

int tpf_glClose (int globaldesc, enum t_glopt options);

GLOBALDESC	The global descriptor returned on a previous call to the tpf_glOpen function, which represents the format-2 global record that is to be closed.

OPTIONS	The options for the close operation. The options indicate whether the global record will have its persistent copy updated before being closed. One of the following options must be specified. **`TPF_GLUPD`** Writes an update to the persistent copy, if applicable, and closes the format-2 global record. **`TPF_GLUPDWT`** Writes an update to the persistent copy, waits for completion (if applicable), and closes the format-2 global record. **`TPF_GLNOU`** Closes the format-2 global record without writing updates to the persistent copy. **`TPF_GLPART`** Updates only the specified part of the global record. This option is specified with either the **TPF_GLUPD** or **TPF_GLUPDWT** option, separated by a + symbol. If this option is specified, the **offset** and **length** parameters ***must*** be included.

Other variants in C:

int tpf_glCntl (int globaldesc, enum t_glopt options);

int tpf_glOpen(char *globalname, enum t_glopt options,

void **globaladdr);

int tpf_glStat(int globaldesc, struct iglst *statinfo);

int tpf_glWrite(int globaldesc, enum t_glopt options);

Entry Conditions:

- R9 must contain the address of the ECB that is being processed.
- If the entry switches from one I-stream to another or from one subsystem user (SSU) to another, any previously opened global records should be closed and reopened to ensure correct addressability to the global.
- The following are requirements when opening, unlocking, or closing a global record that can be synchronized:
  - There should not be any outstanding I/O at the time an update request is issued.
  - The entry must be holding a lock on the global requested to be unlocked or synchronized.
  - You must issue the OPEN and associated UNLOCK or CLOSE in the same segment to ensure the correct protection access. The LOCK should not be carried across Enter macros.

Return Conditions:

Assembler:

R15 contains one of the following values:

GLOBLC_OK

specifies that the requested function completed successfully.

GLOBLC_ERROR

specifies that the requested function completed with errors. A system error might be issued, and the specific error can be determined by using the ERRNOC macro, which will return one of the following values:

ETPFGLOB_BADADDR

indicates that the address specified for the ADDR parameter was NULL.

ETPFGLOB_BADDESC

indicates that the global descriptor specified does not reference a valid open format-2 global record.

ETPFGLOB_BADLENGTH

indicates that the length specified is beyond the end of the specified format-2 global record.

ETPFGLOB_BADNAME

indicates that the global record specified does not indicate the name of a valid format-2 global record.

ETPFGLOB_BADOFFSET

indicates that the offset specified is beyond the end of the specified format-2 global record.

ETPFGLOB_BADOPTIONS

indicates that the options specified are not valid for the function requested.

ETPFGLOB_DELETED

indicates that the format-2 global record has been deleted, so no updates are allowed. The global is closed, but no updates are performed to the persistent copy of the global.

ETPFGLOB_NOHEAP

indicates that there is not enough ECB heap storage to create the format-2 global descriptor.

ETPFGLOB_NOUPDATES

indicates that the format-2 global record cannot be keypointed or synchronized and, therefore, no updates are allowed.

ETPFGLOB_OPEN

indicates that the ECB has the specified format-2 global record open already.

ETPFGLOB_READONLY

indicates that the options specified call for an update to be performed; however, the format-2 global record was opened as read only.

ETPFGLOB_REINIT

indicates that the format-2 global record has been reinitialized and no more actions are allowed.

ETPFGLOB_TIMEOUT

indicates that the format-2 global record synchronization was not completed in the system-controlled time limit. The global record is closed and the persistent copy of the global record has been updated.

ETPFGLOB_UNINIT

indicates that the format-2 global record has not been initialized yet.

FUNC=OPEN

- If successful and OPT=RDONLY or OPT=RDWRITE was specified, the register specified on the DESC parameter contains the global descriptor.
- If successful, the register specified on the ADDR parameter contains the address of the global record requested.
- If the global record specified can be synchronized and OPT=RDWRITE was specified:
  - If an error is returned from the internally generated FIWHC macro while trying to retrieve the format-2 global record, one of the following occurs:
    - If the error is a result of a hold lockout or ZECBL command processing, (CE1SUD=X'81'), it is a user error. System error 064101 is issued and the ECB exits.
    - If the error occurred because of record retrieval problems (for example, a record ID failure occurred), it is related to database corruption. System error 064102 is issued and the subsystem affected is cycled to 1052 state so that you can take immediate corrective action.
  - The main storage copy of the requested global has been refreshed by the lock operation.

FUNC=CNTL

- If successful, the open characteristics of the globals have been changed.
- If OPT=RDWRITE was specified, the global record has been locked for the exclusive use of this ECB.
- If OPT=UNLOCK or OPT=UNLOCKWT was specified, the global record is no longer locked for the exclusive use of this ECB. Additionally, if the global record specified can be synchronized and a previous update was written for this global record, synchronization notification has been sent to all other processors in the tightly coupled and loosely coupled z/TPF complex.

FUNC=WRITE

- If the global record specified can be keypointed, the global record will be keypointed at the next keypoint interval. The global record is still locked by the issuing ECB.
- If the specified global record can be synchronized, the global record has been written to the globals database; however, no synchonization notification has been performed to other processors in the z/TPF complex. The global record is still locked by the issuing ECB.

FUNC=CLOSE

- If OPT=UPDATE or OPT=UPDATEWT was specified, the specified global record was updated in the globals database:
  - If the specified global record can be keypointed, the global record will be keypointed at the next keypoint interval. The global record is unlocked.
  - If the specified global record can be synchronized, the global record has been written to the globals database and synchronization notification has been sent to other processors in the z/TPF complex. The global record is unlocked. If OPT=UPDATEWT was specified, control will not be returned until all other tightly coupled, loosely coupled processors, or both, have acknowledged that the appropriate synchronization has been completed.
- If OPT=NOUPDATE was specified, the global record is closed. If it was opened previously for read and write access, the global record is unlocked.

FUNC=STAT

specifies that the location indicated by the ADDR parameter is initialized with the requested information for the global record specified.

C:

- A value of 0.
- Error return

If unsuccessful, tpf_glClose returns a value of -1, a system error might be issued, and the errno C variable is set to one of the following:

ETPFGLOB_BADDESC

The specified global descriptor does not reference a valid open format-2 global record.

ETPFGLOB_BADLENGTH

The length specified is beyond the end of the specified format-2 global record.

ETPFGLOB_BADOFFSET

The offset specified is beyond the end of the specified format-2 global record.

ETPFGLOB_BADOPTIONS

The options specified are not valid for tpf_glClose.

ETPFGLOB_DELETED

The format-2 global record has been deleted, so no updates are allowed. The global is closed, but no updates are performed to the persistent copy of the global.

ETPFGLOB_NOUPDATES

The format-2 global record cannot be keypointed or synchronized and, therefore, no updates are allowed.

ETPFGLOB_READONLY

The options specified indicate that an update must be performed; however, the format-2 global record was opened as read only.

ETPFGLOB_REINIT

The format-2 global record has been reinitialized, so no updates are allowed. The global is closed, but no updates are performed to the persistent copy of the global.

ETPFGLOB_TIMEOUT

The format-2 global record synchronization did not complete in the system-controlled time limit. The global record is closed and the persistent copy of the global record has been updated.

Exception:

064104	FORMAT-2 GLOBAL SYNC REQUESTED WITH IO PENDING
064108	FORMAT-2 GLOBAL SYNC REQUESTED WHILE SUBSYSTEM USER WAS DORMANT
064107	FORMAT-2 GLOBAL SYNC FILE ERROR
064105	FORMAT-2 GLOBAL SYNC DID NOT COMPLETE IN TIME
064109	FORMAT-2 GLOBAL KEYPOINT FAILED

Programming Consideration:

- Calling this function may cause the ECB to give up control. Therefore, storage protection override must not be turned on when calling this function.

- This function can be used only with z/TPF format-2 globals. To request an update to z/TPF format-1 globals, use the glob_update function.
- Unlike format-1 globals, the names of format-2 global records are the same in C language and basic assembler language (BAL). If the global name is less than 8 characters, it must be padded on the right with blanks.
- When a format-2 global record that was opened originally for read-only access is closed, an update cannot be performed.
- When a format-2 global record that was opened originally for read and write access is closed with no update requested, the global will be unlocked based on the defined attributes of the global. For example, if the global is defined as synchronizable, the equivalent of a synchronization unlock will be issued to unlock the global; if the global is defined as nonsynchronizable, the equivalent of a coruc function will be issued to unlock the global.
- When a format-2 global record that was opened originally for read and write access is closed with an update requested, the global will be updated based on the defined attributes of the global. For example, if the global is defined as synchronizable, the equivalent of a global synchronization will be performed; if the global is defined as keypointable, the z/TPF system will request a keypointing of the global.
- The **TPF_GLUPDWT** option is meaningful only when the global is defined as synchronizable; otherwise, this option will be handled the same as**TPF_GLUPD**.
- The **offset** and **length** parameters are meaningful only when the global is defined as synchronizable; otherwise, these parameters will be ignored.

Example:

Assembler:

The following example requests an update to keypointable format-2 global record _TPFKPT2, and restores storage protection to working storage.

```
GLOBLC FUNC=OPEN,NAME=_TPFKPT2,DESC=R7,ADDR=R4,OPT=RDWRITE
GLMOD F2GLOBP,OVERRIDE=YES
...
GLOBLC FUNC=WRITE,DESC=R7
KEYRC  ,
```

C:

The following example closes a format-2 global record that was opened previously for reading and writing.

```
#include <tpf/c_f2glob.h>
#include <errno.h>
#include <stdio.h>
#include <string.h>
⋮
int   rc;
int   descriptor;
char  *globRecord;
long offset, length;
⋮

descriptor = tpf_glOpen ("_TPFAUX1",
               TPF_GLRDWR,
               &globRecord);

if (descriptor == -1)
```

```
{
      printf ("Got an error on global open - %d", errno);
      exit (0);
}

strcpy (globRecord, "THIS IS JUST A TEST");

rc = tpf_glClose (descriptor,
               (enum t_glopt)(TPF_GLUPD+TPF_GLPART) offset,
               length);
if (rc == -1)
{
 printf ("Got an error on global close - %d", errno);
 exit (0);
}
```

9. Error Handling

9.1 Introduction

- zTPF takes a snapshot of the core, referred to as a ‘Dump’, when an error is detected to aid diagnostics.

- The error can be either detected by the system or an application. When a dump it encountered, it is written to a Real-time tape (RTA/RTL) mounted on the system and a message is sent to the CRAS terminal with information about the dump.

- The dumps written to the Real-time tapes are later post processed on the offline system (MVS) to analyze and determine the cause of the problem.

- When a dump is encountered, all the I-streams are paused to ensure the system core is not altered while being dumped.

- This can take up to a few seconds and hence taking a dump is expensive.

- The dumps can be classified into two categories:
    - Control Dumps
    - Operational Dumps

9.2 Control dumps

- The control dumps refer to errors detected by the control program. Upon encountering a control dump the ECB usually exits but the system continues normal operation.

- However, if a serious condition is detected, a catastrophic dump is taken and a software-IPL (restart) is initiated.

- Examples for catastrophic dump:

CTL-1	Program error in CP
CTL-2	Virtual storage error in CP
CTL-C	Working storage depleted
CTL-11	Long term pool depleted

CTL-21	Hold Table Full

- Examples for non-catastrophic dump:

CTL-3	Protection Exception
CTL-4	Addressing Exception
CTL-10	Application time-out
CTL-22	Find issued on a data level already holding a block.
CTL-35	Attempt to hold same record more than once

9.3 Operational dumps

- The operational dumps are issued by the application (E-type) programs when an application error is encountered.

- An application error could be as a result of invalid data in the database or absence of data in a database etc.

- The application can choose to either continue execution or exit the ECB after taking an operational dump.

- The operational dumps can be coded using one of the two macros, SERRC and the SNAPC.

- The SERRC macro causes all the system activities to be suspended, whereas the SNAPC does not as only a limited amount of data is dumped.

- It is recommended to use SNAPC if the data required to be dumped is less than 32KB.

9.4 Error handling macros

SERRC – Issue system error

SERRC macro causes a storage dump and sends a message to the system operator via a CRAS terminal. It may be used whenever an operational program detects an unexpected condition that requires a dump of main storage as an aid for analysis.

Syntax:

Assembler:

SERRC E/R,errnum,PREFIX=letter,MSG=YES/NO,ECB=YES/NO,LIST=label

E/R	The action to be taken by the system error routine upon completion of the dump. E – EXIT, R – Return to NSI.
errnum	6-digit hexadecimal system error number in the range 000001 to FFFFFF used to identify the dump.
PREFIX	An uppercase alphabetic character that is concatenated with the system error number in the console message and in the dump. Usually U is used in Non-IBM code.
MSG	YES – R0 must contain the address of a user supplied message that is to be appended to the operator notification of the dump.
ECB	YES – Dump the ECB and all working storage mapped in the EVM.
LIST	Specifies the location of one or more LISTC macro calls, which are used to identify additional main storage areas to be dumped.

C:

void serrc_op(enum t_serrc status, int number, const char *msg, void *slist[]);

status The status of the ECB following the dump. This parameter must belong to enumeration type *t_serrc*, which is defined in *tpfapi.h*. Code one of the following defined values:

SERRC_EXIT

Forces the ECB to exit.

SERRC_RETURN

Causes a return to the calling program.

SERRC_CATA

Causes a catastrophic error. When a catastrophic error occurs, the z/TPF system IPLs and returns to 1052 state, unless the system error number is listed in an internal table in program CPSF. If the error number is listed in program CPSF, the system returns to NORM state.

You also can specify whether 31-bit and 64-bit ECB heap buffers are included in the dump by specifying one of the following values *in addition* to one of the previous values:

`SERRC_NO_HEAP`

Does not include any part of the ECB heap in the dump.

`SERRC_ALL_HEAP`

Includes both in-use ECB heap buffers and available ECB heap buffers in the dump.

`SERRC_NO_HEAP`

Exits the ECB and does not include part of the ECB heap in the dump.

`SERRC_ALL_HEAP`

Exits the ECB and includes both in-use ECB heap buffers and available ECB heap buffers in the dump.

`SERRC_NO_HEAP_RETURN`

Returns to the calling program and does not include part of the ECB heap in the dump.

`SERRC_ALL_HEAP_RETURN`

Returns to the calling program and includes both in-use ECB heap buffers and available ECB heap buffers in the dump.

`SERRC_NO_HEAP_CATA`

Causes a catastrophic dump and does not include part of the ECB heap in the dump.

`SERRC_ALL_HEAP_CATA`

Causes a catastrophic dump and includes both in-use ECB heap buffers and available ECB heap buffers in the dump.

number The identification number for the dump. This argument is an integer and should be a unique number ranging from 1 to X'FFFFFF'. This number is prefixed with a U. If you want to control the character that is used as a prefix, use *serrc_op_ext.*

msg A pointer to type *char,* which is a message text string to be displayed at the CRAS console and appended to the dump. This string must be terminated by a \0 and must not exceed 255 characters. Strings longer than 255 characters are truncated at the 255th character. If no message is desired, code the defined term NULL.

slist A pointer to an array of pointers to type *void*, indicating extra areas of storage that are to be displayed in the dump. If no storage list exists, code this parameter as NULL**.** This argument works only for 31-bit addresses. Use the *serrc_op_ext* function with the list parameter for dumping storage involving 64-bit addresses.

Other variants in C:

void	serrc_op_ext(enum t_serrc status, int number, const char *msg,char prefix, struct serrc_list **list);
void	serrc_op_slt(enum t_serrc status, int number, const char *msg,void *slist[], char prefix);

Entry Conditions:

- If MSG=YES is specified, R0 must be loaded with the address of the message.
- The total size of the message is 255 characters including the character count, which must be placed in the first byte.
- If LIST parameter is specified, a valid sequence of LISTC macro must be coded at the specified location.

Return Conditions:

Assembler:

- Control is returned, as specified by the first positional parameter, to NSI when 'R' is specified or to the EXITC macro processing when 'E' is specified.
- All the registers are preserved on return to NSI.
- A storage dump is taken and the system operator is informed of the error via a CRAS set.

C:

void.

Exception: None

Programming Consideration:

All system activity is suspended while the storage dump is written. If the cause of the error is well understood and less than 32KB of data is required to fully document the problem, use the SNAPC macro. The SNAPC macro uses far fewer processor resources; in particular, system activity is not halted during its processing.

Example:

Assembler:

```
      MVI   EBW000,L’MSG
```
Issues OPR-UF85ED2 with message ‘INVALID AIRLINE CODE’, returns to NSI.

```
      MVC   EBW001(L’MSG),MSG
      LA    R0,EBW000
      SERRC R,F85ED2,PREFIX=U,MSG=YES
      .
      .
      MSG   DC    C’INVALID AIRLINE CODE’
```

C:

```
      serrc_op(SERRC_EXIT,0x1234,"cannot find f/a",NULL);
```
Issues OPR-1234 with message ‘Cannot find f/a’ and exit the ECB.

SNAPC – Issue snapshot dump

SNAPC macro, similar to the SERRC, collects diagnostic data to be used for problem determination. If the amount of data needed to analyze is limited, a snapshot dump can be used instead of the SERRC dump to reduce the use of system resources and make the diagnostic data more precise.

Syntax:

Assembler:

SNAPC E/R,errnum,REGS=YES/NO,MSG=label,ECB=YES/NO,LIST=label

E/R	The action to be taken by the system error routine upon completion of the dump. E – EXIT, R – Return to NSI.
errnum	6-digit hexadecimal system error number in the range 000001 to FFFFFF used to identify the dump.
REGS	YES – The general registers are to be included when a snapshot dump is displayed to the system console.
MSG	Specifies the location that contains the message to be appended.
ECB	YES – Dump the ECB and all working storage mapped in the EVM.
LIST	Specifies the location of one or more LISTC macro calls, which are used to identify additional main storage areas to be dumped.

C:

void snapc(int action, int code, const char *msg,struct snapc_list **listc,int regs, int ecb);

action	The action of the ECB after the snapshot dump. This argument is an integer. Code the defined term SNAPC_EXIT to force the ECB to exit, or SNAPC_RETURN to cause a return to the calling program.
code	The identification number for the snapshot dump. This argument is an integer and should be a unique number ranging from 0 to X'7FFFFFFF'.
msg	This argument is a pointer to type *char*, which is a message text string to be displayed at the CRAS console and appended to the dump. The string must be terminated by \0 and must not exceed 255 characters. If no message is desired, code the defined term NULL.

listc	this argument is a pointer to an array of pointers that point to type *struct snapc_list*, indicating areas of storage to be displayed on the dump.
regs	Code the defined term SNAPC_REGS to include the registers in the snapshot dump, or SNAPC_NOREGS to exclude the registers. This argument is an integer.
ecb	Code the defined term SNAPC_NOECB to force the names of the subsystem and subsystem user to that of the basic subsystem.

Other variants in C: None

Entry Conditions:

- If LIST parameter is specified, a valid sequence of LISTC macro must be coded at the specified location.

Return Conditions:

Assembler:

- Control is returned, as specified by the first positional parameter, to NSI when ’R’ is specified or to the EXITC macro processing when ’E’ is specified.
- All the registers are preserved on return to NSI.
- A storage dump is taken and the system operator is informed of the error via a CRAS set.

C:

Void

Exception: None

Programming Consideration:

- The SNAPC service routine will use up to eight 4KB blocks to collect the specified data.
- The SNAPC macro uses far fewer processor resources unlike SERRC; in particular, system activity is not halted during its processing.

Example:

Assembler:

```
SNAPC R,12345,MSG=ERRMSG,REGS=YES,PREFIX=A  Issues a snap dump and returns to
.                                           NSI
.
ERRMSG  EQU    *
LEN     DC     AL1(L’MSG)
MSG     DC     C’INVALID FLIGHT NUMBER’
```

C:

```
     snapc(SNAPC_RETURN,   0x12345,   "CANNOT   FIND   FILE   ADDR",
snapstuff,'U',SNAPC REGS, SNAPC ECB, "C001");
```

Issue a SNAP U12345 with Message ‘Cannot find file addr’ With a return to program by Capturing registers and ECB datas.

LISTC – Dump facility list generator

LISTC macro is used to generate a list of data items for dump processing. The LISTC macro generates constants that identify the name, size, and location of the items that need to be included in the dump when the LIST parameter is supplied.

Syntax:

Assembler:

LISTC NAME=field1, TAG GROUP, LEVEL=Dn, END

TAG GROUP TAG=field2, LEN=sd, INDIR=YES or NO

NAME	Specifies the name of the field to appear in the dump.
TAG	Specifies the location of the data to be dumped.
LEN	Specifies the length of the field to be dumped. If omitted, it defaults to the length of the field specified in the TAG parameter.
INDIR	YES – Indicates that the TAG parameter specifies the location that contains the 4-byte address of the data to be dumped. NO – Indicates that the TAG parameter specifies the direct location the data to be dumped. The default is NO.
LEVEL	Used in lieu of the TAG, LEN and INDIR parameters to identify a data area to be dumped. It identifies the data level, where the storage address (CE1CRx) and length (CE1CCx) of the data to be dumped can be found.
END	This parameter is mutually exclusive with all other parameters, and marks the end of the list of items specified by the LIST option. All LISTC lists must end with the END option.

C: None

Other variants in C:

Entry Conditions: None

Return Conditions:

Assembler: None

Exception: None

Programming Consideration:

- The LISTC macro should be in the data constant section of the program and should never be executed as it generates data and NOT executable code.

Example:

Assembler:

```
SNAPC R,12345,PREFIX=A,LIST=DAT2DMP
.                    Dumps 104 bytes starting from EBW000 and data level D0.


.
DAT2DMP EQU*
LISTC NAME=EBWAREA,TAG=EBW000,LEN=104
LISTC NAME=LVLZERO,LEVEL=D0
LISTC END
```

Q1. Which one is best SERRC or SNAPC and why?

10. Magnetic Tape

10.1 Introduction

- In a zTPF system, the magnetic tapes are used for both online and offline processing.

- The tapes are used as both Input as well as Output medium. It is the prime medium for passing data to and receiving data from other systems such as MVS.

- The tapes are identified by 3-character name and it has to be mounted by a system operator on a tape drive attached to the zTPF system.

- The tapes are classified into two categories:

  - Real-time tapes
  - General tapes

10.2 Real-time tapes

- The Real-time tapes are Write-Only tapes that are available to all entries in the system.

  - **tourc** – Write record, release buffer block
  - **toutc** – Write record, retain buffer block

- Records are written on these tapes in the order in which the control program receives TOUTC and TOURC macros. Each record is identified by the subsystem and time stamped.

- The Real-time tapes are used to:

  - Log transactions
  - Collect dynamic system information
  - Record main storage dumps resulting from system errors

- The name of the real-time tapes begins with characters RT. Example, RTA and RTL. RTA is the primary real-time tape and RTL is the logging real-time tape. RTA and RTL must always be mounted in the system.

10.3 General tapes

- General tape control is process based. All entry control blocks (ECBs) that belong to the same process (that is, threads) share control and access to the same tapes.
- General tapes are I/O tapes used for application programming. They allow the application program to write to consecutive files and read them in logical sequence.
- There are two groups of API functions for managing general tapes: basic general tape functions and high-level general tape functions.
    - A basic general tape function performs a single tape function and gives an ECB absolute control over a tape. (However, a tape can be shared between ECBs by assigning and reserving a tape in the appropriate sequence.) The basic general tape functions are:

      tasnc Assign tape to process
      tbspc Backspace tape
      tclsc Close a general tape
      tdspc Display tape status
      tdspc_q Display tape queue length
      topnc Open tape
      tprdc Read tape record
      trewc Rewind tape
      trsvc Reserve tape for other processes
      tsync Flush tape buffer
      twrtc Write tape record.

    - A high-level general tape function performs multiple tape functions from the set of basic general tape functions and allows all ECBs to share a tape. The high-level general tape functions are:
      tape_close Close a general tape
      tape_cntl Tape position control
      tape_open Open tape
      tape_read Read a record
      tape_write Write a record

- The tape status table (TSTB) is used to control tape operations. This table contains hardware addresses, device status, symbolic name assignments, and queuing and chaining information..

10.4 Tape macros

TOURC – Write a Real-Time Tape Record and Release Core Block

This general macro writes a record contained in a core block to a real-time tape. It will also return the core block to the appropriate pool and make it unavailable to the operational program.

Syntax:

Assembler:

TOURC NAME=ccc or R(n),LEVEL=level

NAME	*ccc* specifies a 3-character string representing a symbolic real-time tape name. The first 2 characters must be alphabetic, and the third character must be alphabetic or numeric. **(R*n*)** specifies the number of a register containing a pointer to the symbolic real-time tape name. *n* must be a decimal number from 0 – 7, 14, or 15.
level	specifies a symbolic data level (D0–DF).

C:

void tourc(const char *name, enum t_lvl level);

Name	This argument is a pointer to type char, which must be a 3-character string identifying the tape to be written to. This function can only be called for a real-time tape.
Level	One of 16 possible values representing a valid data level from the enumeration type t_lvl, expressed as D*X*, where *X* represents the hexadecimal number of the level (0–F). The working storage block on this level is the record to be written to tape.

Other variants in C: None

Entry Conditions:

- A core block must be held by the ECB on the data level or the DECB specified by this macro.

Return Conditions:

Assembler:

- Control is returned to the next sequential instruction.
- The contents of R14 and R15 are unknown. Contents of all other registers are preserved across this macro call.
- The core block containing the data record is no longer available to the operational program.

- The file address reference word (FARW) on the data level or DECB specified by this macro is unchanged.
- The core block reference word (CBRW) on the data level or DECB specified by this macro is updated to indicate that the storage block is no longer held by the ECB.

C:

Void

Exception:

CTL 59	NO BLK ATTACHED FOR TWRTC/TOURC
CTL 5C	TAPE MACRO USED PRIOR TO TAPE RESTART COMPLETION

Programming Consideration:

- The status of the Write operation can never be determined by the operational program.
- The record written to tape will have 16 bytes of appended data containing the subsystem name, subsystem user name, and value of the TOD clock.
  Note: The value of the TOD clock will be the time at which the macro was issued.
- If the tape is mounted on a buffered device, the operation is performed in buffered mode.
- If this macro is issued prior to the end of tape restart, the ECB is exited and a system error issued.

Example:

Assembler:

```
TOURC NAME=ABC,LEVEL=D5
```
Write record to tape

C:

```
#include <tpf/tpftape.h>
  tourc("RTA",D5);
```
Write record to tape

TOUTC – Write a Real-Time Tape Record

This general macro will write a record contained in a main storage to a real-time tape. The record can be in a permanent storage resident area, a working storage block, or the fixed work area in the ECB.

Syntax:

Assembler:

TOUTC NAME=ccc or R(n), LEVEL=Dx,BUF=No or Yes

NAME Specifies the symbolic real-time tape name. It can be:
ccc
A 3-character string representing a symbolic real-time tape name. The first 2 characters must be alphabetic and the third character must be alphabetic or numeric.
(R*n*)
The number of a register containing a pointer to the symbolic real-time tapename. *n* must be a decimal number from 0 through 7, 14, or 15.

level A symbolic data level (D0–DF) must be specified.

BUF An optional keyword parameter can be specified indicating the output mode to be used when writing to buffered devices.
YES
Buffered mode is to be used.
NO
Tape Write Immediate (TWI) mode is to be used.
If omitted, a default of BUF=NO is assumed. This parameter has meaning only when the tape is mounted on a buffered device. It is ignored when the tape is mounted on a non-buffered device.

C:

void toutc(const char *name, enum t_lvl level, int bufmode);

Name This argument is a pointer to type char, which must be a 3-character string identifying the tape to be written to. This function can only be called for a real-time tape.

Level One of 16 possible values representing a valid data level from the enumeration type t_lvl, expressed as D*X*, where *X* represents the hexadecimal number of the level (0–F). The working storage block on this level is the record to be written to tape.

Bufmode Whether the tape device being written to is written in buffered mode or write immediate mode. Use the defined terms **NOBUFF** to denote no buffering, or **BUFFERED** to denote buffered write mode.

Other variants in C: None

Entry Conditions:

- The first 4 bytes of the file address reference word (FARW) of the data level specified by this macro must contain the address of the data. The record to be written must be above X'1000'. The last 2 bytes of the FARW must contain the byte count of the record. This byte count must be in the range 1–32 752.

Return Conditions:

Assembler:

- Control is returned to the next sequential instruction.
- The contents of R14 and R15 are unknown. The contents of all other registers are preserved across this macro call.
- The status of the Write operation is unknown. The FARW on the data level specified by this macro is unchanged.

C:

Void

Exception:

CTL 4F	MINIMUM BYTE COUNT NOT MET
CTL 5C	TAPE MACRO USED PRIOR TO TAPE RESTART COMPLETION

Programming Consideration:

Tape data transfer depends on various conditions:

- For a blocked tape, a WAITC ensures that the data has been transferred to the blocking buffer. This is true whether the tape is mounted on a buffered device or an non-buffered device.
- For an unblocked tape mounted on a buffered device that is operating in buffered mode, a WAITC guarantees that the data has been written to the control unit buffer.
  - For any other unblocked tape, a WAITC guarantees that the data has been written to the tape.
- The record written to tape will have 16 bytes of appended data containing the subsystem name, subsystem user name, and value of the time-of-day clock.
  Note: The value of the time-of-day clock will be the time at which the macro was issued.

Example:

Assembler:

```
TOUTC NAME=ABC,LEVEL=D5,BUF=NO      Write record to tape
```

C:

```
toutc("RTA",D9,BUFFERED);     Write record to tape
if (waitc())
{
  serrc_op(SERRC_EXIT,0x1234,"ERROR ON RTA TAPE",NULL) ;
}
```

TOPNC – Open a General Tape

This general macro makes the specified general tape available to the operational program.

Syntax:

Assembler:

TOPNC NAME=ccc or R(n),STATUS=I/O,BUF=Yes or No

NAME	Specifies the symbolic real-time tape name. It can be: *ccc* A 3-character string representing a symbolic real-time tape name. The first 2 characters must be alphabetic and the third character must be alphabetic or numeric. **(R*n*)** The number of a register containing a pointer to the symbolic real-time tapename. *n* must be a decimal number from 0 through 7, 14, or 15.
Status	The input or output status must be specified. This parameter must be coded as either I or O.
BUF	An optional keyword parameter can be specified indicating the output mode to be used when writing to buffered devices. **YES** Buffered mode is to be used. **NO** Tape Write Immediate (TWI) mode is to be used. If omitted, a default of BUF=NO is assumed. This parameter has meaning only when the tape is mounted on a buffered device. It is ignored when the tape is mounted on a non-buffered device.

C:

void topnc(const char *name, int io, int bufmode);

Name	This argument is a pointer to type char, which must be a 3-character string identifying the tape to be written to. This function can only be called for a real-time tape.
Io	This argument is treated as an integer describing whether the tape is to be read (input) or written (output). Use the defined terms **INPUT** to denote an input tape or **OUTPUT** to denote an output tape.
bufmode	Whether the tape device being written to is written in buffered mode or write immediate mode. Use the defined terms **NOBUFF** to denote no buffering, or **BUFFERED** to denote buffered write mode.

Other variants in C: None

Entry Conditions:

- The general tape specified by this macro must not be open when this macro is issued.

Return Conditions:

Assembler:

- The contents of R14 and R15 are unknown. The contents of all other registers are preserved across this macro call.
- Control is returned to the operational program when the specified tape is ready for use.
- The positioning of the specified tape is unchanged.

C:

- The specified tape is positioned at the first record, and the tape has been assigned to the issuing ECB.

Exception:

CTL 57	TOPNC PARMS NON COMPATIBLE WITH TSTB
CTL 5C	TAPE MACRO USED PRIOR TO TAPE RESTART COMPLETION
CTL 51	TAPE *tapename* NOT IN TSTB
CTL 54	TAPE *tapename* ALREADY OPEN
CTL 52	TASNC ISSUED BUT TAPE NOT OPEN
CTL 53	TAPE *tapename* NOT HELD

Programming Consideration:

- TCLSC macro or TRSVC macro must be issued for all open tapes prior to issuing an EXITC macro.
- If the tape is mounted on a non-buffered device, the buffered mode keyword parameter (BUF) is ignored.
- If the tape is opened as an output tape and is mounted in blocked mode, the buffered mode keyword parameter (BUF) causes the appropriate indicator to be set in the tape status table entry. However, this indicator will be ignored for subsequent writes to the tape, and all write operations will be performed in buffered mode.

Example:

Assembler:

```
TOPNC NAME=ABC,LEVEL=D5,BUF=NO
```
Open a general tape

C:

```
topnc("VPH",INPUT,NOBUFF);
```
Open a general tape

TCLSC – Close a General Tape

This macro returns a specified general tape to the system. This function closes a general tape, forcing a logical dismount from the system and causing the tape to be physically rewound and unloaded from the tape drive. The named tape must be physically mounted, logically open, and assigned to the issuing entry control block (ECB).

Syntax:

Assembler:

TCLSC NAME=ccc or R(n),D= Yes or No,R= Yes or No,U= Yes or No, EXIT= Yes or No

NAME Specifies the symbolic real-time tape name. It can be:
ccc
A 3-character string representing a symbolic real-time tape name. The first 2 characters must be alphabetic and the third character must be alphabetic or numeric.
(R*n*)
The number of a register containing a pointer to the symbolic real-time tapename. *n* must be a decimal number from 0 through 7, 14, or 15.

D=YES|NO
is an optional parameter that can be coded to indicate if a dismount is to be performed for this tape, which would remove the tape name from the tape status table. If specified, this parameter must be coded as YES or NO. If omitted, a default of YES is assumed.
R=YES|NO
is an optional parameter that can be coded to indicate whether a rewind is to be performed for this tape. If specified, this parameter must be coded as YES or NO. If omitted, a default of YES is assumed.
U=YES|NO
is an optional parameter that can be coded to indicate whether an unload is to be performed for this tape. If specified, this parameter must be coded as YES or NO. If omitted, a default of YES is assumed.
EXIT=YES
is a special interface for EXITC processing. This interface will allow tape close processing to force the tape to be closed and bypass any resulting dumps. This parameter is only valid for the EXITC service routine.

C:

void tclsc(const char *name);

Name This argument is a pointer to type char, which must be a 3-character string identifying the tape to be written to. This function can only be called for a real-time tape.

Other variants in C: None

Entry Conditions:

- The general tape specified by this macro must be open when this macro is issued.

Return Conditions:

Assembler:

- Control is returned to the next sequential instruction.
- The contents of R14 and R15 are unknown. The contents of all other registers are preserved across this macro call.

C:

Void

Exception:

CTL 53	TAPE *tapename* NOT HELD

Programming Consideration:

- If the rewind parameter (R) is coded as NO, the tape is closed and positioned before the tape mark which precedes the trailer labels on the current volume of the dataset. If the rewind parameter (R) is coded as YES and the unload parameter (U) is coded as NO, the tape is closed and positioned after the tape mark which follows the header labels on the first volume of the dataset.
- A TCLSC macro or TRSVC macro must be issued for all open tapes prior to issuing an EXITC macro.
- A TCLSC macro performs the equivalent of a TSYNC macro before closing the tape to ensure that all data from a tape buffer is physically written to the tape.

Example:

Assembler:

```
TCLSC NAME=ABC
```
Close a general tape

C:

```
tclsc("VPH");
```
Close a general tape

TWRTC – Write a General Tape Record

This general macro writes a record from a core block to the specified general tape.

The block of storage containing the data record is detached from the entry control block (ECB) and returned to the appropriate pool.

Syntax:

Assembler:

TWTRC NAME=ccc or R(n), LEVEL/DECB=Dx/decbaddr

NAME	Specifies the symbolic real-time tape name. It can be: *ccc* A 3-character string representing a symbolic real-time tape name. The first 2 characters must be alphabetic and the third character must be alphabetic or numeric. **(R*n*)** The number of a register containing a pointer to the symbolic real-time tapename. *n* must be a decimal number from 0 through 7, 14, or 15.
Level/decb	A symbolic data level (D0–DF) or the decb address must be specified.

C:

void twrtc(const char *name, enum t_lvl level);

Name	This argument is a pointer to type char, which must be a 3-character string identifying the tape to be written to. This function can only be called for a real-time tape.
Level	One of 16 possible values representing a valid data level from the enumeration type t_lvl, expressed as D*X*, where *X* represents the hexadecimal number of the level (0–F). The working storage block on this level is the record to be written to tape.

Other variants in C: None

Entry Conditions:

- The general tape specified by this macro must be open when this macro is issued.

Return Conditions:

Assembler:

- Control is returned to the next sequential instruction.
- The contents of R14 and R15 are unknown. The contents of all other registers are preserved across this macro call.
- The file address reference word (FARW) on the data level or DECB specified is unchanged.
- The core block reference word (CBRW) on the data level or DECB is updated to indicate that the storage block is no longer held by the ECB.
- When a condition such as End-of-Volume (EOV) or permanent error occurs, a tape switch will automatically occur. For the TWRTC macro, AUTO=YES is implied.

C:

Void

Exception:

CTL 59	NO BLK ATTACHED FOR TWRTC/TOURC
CTL 53	TAPE *tapename* NOT HELD

Programming Consideration:

- The status of the Write operation can never be determined by the operational program.
- The contents of the entire storage block are written to tape. When writing to an unblocked general tape mounted on a buffered device, the mode of operation (buffered or Tape Write Immediate) is determined by the setting of the tertiary status byte in the tape status table.

Example:

Assembler:

```
TWRTC NAME=ABC,LEVEL=D5
```
Write record to tape

C:

```
twrtc("VPH",D3);
```
Write record to tape

TDSPC – Display tape status

Use this general macro to provide the status of the specified tape.

Syntax:

Assembler:

TDSPC NAME=ccc or R(n), STATUS=A/S,LEVEL=Dn,FORMAT=TPIND/QUEUE

NAME Specifies the symbolic real-time tape name. It can be:
ccc
A 3-character string representing a symbolic real-time tape name. The first 2 characters must be alphabetic and the third character must be alphabetic or numeric.
(R*n*)
The number of a register containing a pointer to the symbolic real-time tapename. *n* must be a decimal number from 0 through 7, 14, or 15.

Status A=Active, S=Standby tape
Format TPIND=returns the status of the tape, QUEUE=returns module queue length of tape

level A symbolic data level (D0–DF) must be specified.

C:

Struct tpstat *tdspc(const char *name,char type, enum t_lvl level);

Name This argument is a pointer to type char, which must be a 3-character string identifying the tape to be written to. This function can only be called for a real-time tape.

Type=whether the active or standby tape status is to be checked

Level One of 16 possible values representing a valid data level from the enumeration type t_lvl, expressed as D*X*, where *X* represents the hexadecimal number of the level (0–F). The working storage block on this level is the record to be written to tape.

Other variants in C: None

Entry Conditions:

- R9 must contain the address of the ECB that is being processed

Return Conditions:

Assembler:

- Control is returned to the next sequential instruction (NSI).
- The contents of R14 is unknown. R15 will point to the Tape Status Table entry of the tape name specified. The contents of all other registers are preserved across this macro call.
- Certain information from the tape status table is placed in the file address reference word (FARW) on the level specified by this macro

C:

Pointer to structure `tpstat` (defined in `tpftape.h`) containing tape status. The indicated tape need not be assigned to the issuing ECB.

Exception:

CTL 4A	MAXIMUM BYTE COUNT EXCEEDED
CTL 49	INVALID BLOCKED TAPE OPERATION
CTL 53	INVALID BLOCKED TAPE OPERATION

Programming Consideration:

- If the tape status table does not contain an entry for the specified tape, the FARW is set to zeros.
- Real-time tape names can be given aliases, or tape pseudonames. These pseudonames are defined using RTMAP definitions in CEFZ. This mechanism provides a means for mapping many names to just a few tape devices..

Example:

Assembler:

```
     TDSPC NAME=(R6),STATUS = A,LEVEL=D3
```
Check tape status

C:

```
#include <tpf/tpftape.h>
struct tpstat *status;

if ((status = tdspc("VPH", ACTIVE, D0)) == NULL)
  {
   serrc_op(SERRC_EXIT,0x1234,"VPH TAPE NOT MOUNTED",NULL) ;
  }
```

TBSPC – Backspace General Tape and Wait

Use this general macro to move the specified general tape backward over a specified number of physical blocks.

Syntax:

Assembler:

TBSPC NAME=ccc or R(n), LEVEL=Dx,F=Yes or No

NAME Specifies the symbolic real-time tape name. It can be:
ccc
A 3-character string representing a symbolic real-time tape name. The first 2 characters must be alphabetic and the third character must be alphabetic or numeric.
(R*n*)
The number of a register containing a pointer to the symbolic real-time tapename. *n* must be a decimal number from 0 through 7, 14, or 15.

level A symbolic data level (D0–DF) must be specified.

F is an optional keyword parameter that can be specified indicating whether fallback to a previous volume is to be allowed when processing multivolume files. This will occur when an attempt is made to backspace a tape past the load point.

YES

specifies that fallback is to be allowed.

NO

specifies that fallback is to be inhibited.

If omitted, a default of YES is assumed.

C:

int tbspc(const char *name, enum t_lvl level, int fallback);

Name This argument is a pointer to type char, which must be a 3-character string identifying the tape to be written to. This function can only be called for a real-time tape.

Level One of 16 possible values representing a valid data level from the enumeration type t_lvl, expressed as D*X*, where *X* represents the hexadecimal number of the level (0–F). The working storage block on this level is the record to be written to tape.

fallback This argument applies when operating a on a multivolume tape file. If **FALLBACK** is specified, the previous volume is mounted if the load point was reached on the current volume and more records exist to backspace. If **NO_FALLBACK** is specified, backspacing stops at the load point with no fallback to the previous volume.

Other variants in C: None

Entry Conditions:

- R9 must contain the address of the ECB being processed.
- The general tape specified by this macro must be open when this macro is issued.
- The number of physical blocks to be backspaced must be contained in the low-order 2 bytes of the file address reference word (FARW) of the data level specified by this macro. This number cannot be zero and cannot exceed 65 535. Attempts to backspace more than 65 535 records will backspace the number of records equal to the value entered mod 64 K.

Return Conditions:

Assembler:

- If no I/O hardware errors or unusual conditions have occurred, control is returned to the next sequential instruction (NSI). Otherwise, control is returned to the error routine.
- The contents of R14 and R15 are unknown. The contents of all other registers are preserved across this macro call.
- All pending I/O operations (including this request) are complete for this ECB.
- For I/O hardware errors, the system error routine has taken a storage dump and informed CRAS.

C:

- Integer value of zero if the operation is successful.
- The value of CE1SDx is returned if an I/O error occurs.

Exception:

CTL 49	INVALID BLOCKED TAPE OPERATION
CTL 53	TAPE *tapename* NOT HELD

Programming Consideration:

- Specifying invalid data level results in a system error with exit.
- The tape must already have been assigned to the issuing ECB before this function is called.
- This function calls the equivalent of waitc.
- The number of physical blocks to be backspaced must be from 1 to 65 535. Attempts to backspace more than 65 535 records will backspace the number of records equal to the value entered mod 64 K.
- If you issue the tape functions in a multithreaded environment, ensure that the functions are issued in the required sequence.

Example:

Assembler:

`TBSPC  NAME=ABC,LEVEL=D5` *Moves specified general tape backwards over a* *specified number of physical blocks*

C:

```
  if  (tbspc("VPH",D9,FALLBACK))
```
backspaces the VPH tape by the specified number of records.

```
  {
   serrc_op(SERRC_EXIT,0x1234,"ERROR BACKSPACING VPH TAPE",NULL) ;
  }
```

TREWC – Rewind General Tape and Wait

This general macro positions the specified general tape at the beginning of its initial data record.

Syntax:

Assembler:

TREWC NAME=ccc or R(n),F=Yes or No

NAME	Specifies the symbolic real-time tape name. It can be: *ccc* A 3-character string representing a symbolic real-time tape name. The first 2 characters must be alphabetic and the third character must be alphabetic or numeric. **(R*n*)** The number of a register containing a pointer to the symbolic real-time tapename. *n* must be a decimal number from 0 through 7, 14, or 15.
F	is an optional keyword parameter that can be specified to indicate whether fallback to the first volume of a multivolume file is to be allowed. **YES** indicates that fallback is to be allowed. **NO** indicates that fallback is to be inhibited. If omitted, a default of YES is assumed.

C:

int trewc(const char *name, int fallback);

Name	This argument is a pointer to type char, which must be a 3-character string identifying the tape to be written to. This function can only be called for a real-time tape.
fallback	This argument applies when operating a multi-volume tape file. If **FALLBACK** is specified, the previous volume is mounted when load point is reached on the current volume. If **NO_FALLBACK** is specified, rewinding stops at the load point with no fallback to the previous volume.

Other variants in C: None

Entry Conditions:
- R9 must contain the address of the ECB being processed.
- The general tape specified by this macro must be open when this macro is issued.

Return Conditions:

Assembler:

- If no I/O hardware errors or unusual conditions have occurred, control is returned to the next sequential instruction (NSI). Otherwise control is transferred to the address specified by this macro.

- The contents of R14 and R15 are unknown. The contents of all other registers are preserved across this macro call.
- All pending I/O operations, including this request, are complete for this ECB.
- For I/O hardware errors, the system error routine has taken a storage dump and informed CRAS.

C:

- Integer value of zero indicating successful completion.
- The nonzero integer value in CE1SUG is returned to the caller.

Exception:

CTL 53	TAPE *tapename* NOT HELD

Programming Consideration:

- This macro can be executed on any **I**-stream.
- Both keyword and positional parameters may be used in the same macro call. This practice is not recommended.
- If any I/O hardware errors have occurred, detailed information has been stored in the ECB.
- If you issue the tape macros in a multithreaded environment, ensure that the macros are issued in the required sequence.

Example:

Assembler:

`TREWC NAME=ABC` *rewinds the current physical volume of the VPH tape*.

C:

`trewc("VPH",NO_FALLBACK);` *rewinds the current physical volume of VPH tape.*

TRSVC – Reserve General Tape

This general macro reserves the specified general tape for use by some future ECB. The tape will retain its current positioning until the future ECB takes control with a TASNC macro.

Syntax:

Assembler:

TRSVC NAME=ccc or R(n)

NAME	Specifies the symbolic real-time tape name. It can be: *ccc* A 3-character string representing a symbolic real-time tape name. The first 2 characters must be alphabetic and the third character must be alphabetic or numeric. **(R*n*)** The number of a register containing a pointer to the symbolic real-time tapename. *n* must be a decimal number from 0 through 7, 14, or 15.

C:

void trsvc(const char *name);

Name	This argument is a pointer to type char, which must be a 3-character string identifying the tape to be written to. This function can only be called for a real-time tape.

Other variants in C: None

Entry Conditions:

- R9 must contain the address of the ECB being processed.
- The general tape specified by this macro must be open when this macro is issued.

Return Conditions:

Assembler:

- Control is returned to the next sequential instruction.
- The contents of R14 and R15 are unknown. The contents of all other registers are preserved across this macro call.

C:

Void

Exception:

CTL 53	TAPE *tapename* NOT HELD

Programming Consideration:

- This macro can be executed on any **I**-stream.
- Both keyword and positional parameters may be used in the same macro call.
- Only a TASNC macro may be issued to a general tape after a TRSVC macro is issued.
- A TCLSC macro or TRSVC macro must be issued for all open tapes prior to issuing an

EXITC macro.
- If you issue the tape macros in a multithreaded environment, ensure that the macros are issued in the required sequence

Example:

Assembler:

```
TRSVC NAME=ABC
```
Opens the tape ABC and reserves it.

C:

```
trsvc("VPH");
```
Opens the tape ABC and reserves it.

TASNC – Assigns a General Tape

Use this general macro to assign the specified general tape to the process that issues the TASNC macro.

Syntax:

Assembler:

TASNC NAME=ccc or R(n)

NAME Specifies the symbolic real-time tape name. It can be:
ccc
A 3-character string representing a symbolic real-time tape name. The first 2 characters must be alphabetic and the third character must be alphabetic or numeric.
(R*n*)
The number of a register containing a pointer to the symbolic real-time tapename. *n* must be a decimal number from 0 through 7, 14, or 15.

C:

void tasnc(const char *name);

Name This argument is a pointer to type char, which must be a 3-character string identifying the tape to be written to. This function can only be called for a real-time tape.

Other variants in C: None

Entry Conditions:

- R9 must contain the address of the ECB being processed.
- The general tape specified by this macro must be open when this macro is issued.

Return Conditions:

Assembler:

- Control is returned to the next sequential instruction.
- The contents of R14 and R15 are unknown. The contents of all other registers are preserved across this macro call.
- The tape position is unchanged.

C:

Void

Exception:

CTL 55	TASNC ISSUED – TAPE NOT AVAILABLE
CTL 5C	TAPE MACRO USED PRIOR TO TAPE RESTART COMPLETION
CTL 52	TASNC ISSUED BUT TAPE NOT OPEN

CTL 51	TAPE *tapename* NOT IN TSTB
CTL 53	TAPE *tapename* NOT HELD

Programming Consideration:

- This macro can be executed on any **I**-stream.
- Both keyword and positional parameters may be used in the same macro call.
- If the specified tape was not reserved before this macro was issued, processing of the entry is suspended until the tape is placed in reserved status. This implied wait can cause the entry to give up control of the system.
- If this macro is issued prior to the end of tape restart, the ECB is exited and a system error issued.
- If you issue the tape macros in a multithreaded environment, ensure that the macros are issued in the required sequence.

Example:

Assembler:

```
TASNC NAME=ABC
```

Assigns a general tape ABC.

C:

```
tasnc("VPH");
```

Assigns a general tape VPH.

11. System Utilities

11.1 File capture and restore

The TPF online database mostly contains business critical data, loss of which could lead to a major disaster. Hence a back-up is necessary for disaster recovery. The File Capture and Restore facility serves this purpose.

Capture

The File Capture (or simply Capture) is a system utility that copies files from online database into magnetic tapes. It is usually performed when the system activity is low.

Exception Logging

When the File Capture is in progress, updates may have been done to the records that are already captured. To ensure that the back data is accurate, TPF keeps track of the updated records and logs them into separate tape called the Exception tape.

Restore

The restore function combines the output of Capture and Exception Logging functions and reconstructs the database either partially or completely based on the request.

11.2 Recoup

Recoup is a system utility that checks the file structure and pool indexes for any discrepancy and provides an error summary for analysis and corrective actions. The discrepancy may occur as a result of:

- *Bug in an application program* - For instance, an application program could have released a long-term pool record, but failed to clear its reference from a fixed record or it could have cleared the reference, but failed to release the record. The former is referred to as ‘Erroneously available file address’ and the later as ‘Lost address’.

- *Action by system programs during a system restart* - At restart, the system programs marks a predefined number of pool records as 'in-use' to ensure integrity of the database during the restart procedure.

- In either case, records are marked as 'in use' when in actual, they are not. If left unattended, this could cause serious depletion of available pool records.

- The recoup process closely interfaces with the application environment to identify the pool records that are really in use and return the records that are not and considered lost.

- It also provides an error summary which helps in identifying and correcting erroneous application program.

11.3 Pool Directory update

- The Pool Directory Update (or simply, PDU) is the process used to return long-term pool records available for reuse by the system.

- Every time an application issues a macro to release a pool record or chain of pool records, the address of the record or records is written to the Real-time tape.

- These tapes are then processed in the offline system to provide a list of records that can be returned to the system.

11.4 System Utility Macros

CS – Compare and swap 4-byte values

This function is used to compare and swap 4-byte values in a storage location that is accessed by multiple I-streams.

Syntax:

Assembler: None

C:

int cs(cs_t *oldptr, cs_t *curptr, cs_t newvalue);

Oldptr	The address of a 4-byte value that is compared to the value pointed to by the **curptr** parameter.
Curptr	The address of a 4-byte value that is to be replaced by the **newvalue** parameter.
Newvalue	The value to be copied to the address specified by the **curptr** parameter.

Other variants in C: None

Entry Conditions: None

Return Conditions:

Assembler: None

C:

- The **newvalue** parameter value is copied to the storage location pointed to by the **curptr** parameter. A value of 0 is returned to the caller.

- A value of 1 is returned if the 4-byte values pointed to by the **oldptr** and **curptr** parameters are not equal. The value pointed to by the **curptr** parameter is copied into the storage location pointed to by the **oldptr** parameter.

Exception: None

Programming Consideration:

- The value pointed to by the **curptr** parameter must be on a 4-byte boundary.
- Use the tpf_serialized_update4 function instead of the cs function for simple increments or decrements.

Example:

Assembler:

C:

The following example shows that 1 is added to the value in a storage location.

```
#include <tpf/cmpswp.h>
int *x;             /* Ensure that the variable is
                    initialized before the do loop. */
int y;
int z;
⋮
do {
     y = *x;       /* y contains the old value of x*/
     z = y+1;      /* z contains x+1 */

   } while (cs((cs_t *)&y, (cs_t *)x, (cs_t)z) != 0);
                   /* Perform compare and swap. If it fails,
                   try again */
```

CSG – Compare and swap 8-byte values

This function is used to compare and swap 8-byte values in a storage location that is accessed by multiple I-streams.

Syntax:

Assembler: None

C:

int csg(csg_t *oldptr, csg_t *curptr, csg_t newvalue);

Oldptr	The address of an 8-byte value that is compared to the value pointed to by the **curptr** parameter.
Curptr	The address of an 8-byte value that is to be replaced by the **newvalue** parameter.
Newvalue	The value to be copied to the address specified by the **curptr** parameter.

Other variants in C: None

Entry Conditions: None

Return Conditions:

Assembler: None

C:

- The **newvalue** parameter value is copied to the storage location pointed to by the **curptr** parameter. A value of 0 is returned to the caller.
- A value of 1 is returned if the 8-byte values pointed to by the **oldptr** and **curptr** parameters are not equal. The value pointed to by the **curptr** parameter is copied into the storage location pointed to by the **oldptr** parameter.

Exception: None

Programming Consideration:

- The value pointed to by the **curptr** parameter must be on an 8-byte boundary.
- Use the tpf_serialized_update8 function instead of the cds and csg functions for simple increments or decrements.

Example:

Assembler:

C:

The following example shows that 1 is added to the value in a storage location.

```
#include <tpf/cwpswp.h>
long *x;          /* Ensure that the variable is
                   initialized before the do loop. */
long y;
long z;
⋮
do {
    y = *x;       /* y contains the old value of x*/
    z = y+1;      /* z contains x+1 */
   } while (csg((csg_t *)&y, (csg_t *)x, (csg_t)z) != 0);
                 /* Perform compare and swap. If it fails,
                    try again */
```

CDS – Compare double and swap 8-byte values

This function is used to compare and swap 8-byte values in a storage location that is accessed by multiple I-streams.

Syntax:

Assembler: None

C:

int cds(cds_t *oldptr, cds_t *curptr, cds_t newvalue);

Oldptr	The address of an 8-byte value that is compared to the value pointed to by the **curptr** parameter.
Curptr	The address of an 8-byte value that is to be replaced by the **newvalue** parameter.
Newvalue	The value to be copied to the address specified by the **curptr** parameter.

Other variants in C: None

Entry Conditions: None

Return Conditions:

Assembler: None

C:

- The **newvalue** parameter value is copied to the storage location pointed to by the **curptr** parameter. A value of 0 is returned to the caller.

- A value of 1 is returned if the 8-byte values pointed to by the **oldptr** and **curptr** parameters are not equal. The value pointed to by the **curptr** parameter is copied into the storage location pointed to by the **oldptr** parameter.

Exception: None

Programming Consideration:

- The value pointed to by the **curptr** parameter must be on an 8-byte boundary.
- Use the tpf_serialized_update8 function instead of the cds and csg functions for simple increments or decrements.

Example:

Assembler: None

C:

The following example shows that 1 is added to the first word in storage and 1 is subtracted from the second word in storage.

```
#include <tpf/cmpswp.h>
cds_t *x;             /* Ensure that the variable is
                      initialized before the do loop. */
cds_t  y;
cds_t  z;
      ⋮
do
{
 y = *x;          /* y contains the old value of x */
 z.double_word.first_word = y.double_word.first_word + 1;
 z.double_word.second_word = y.double_word.second_word - 1;
                  /* z contains the new value */
} while (cds(&y, x, z) != 0);
                 /* Perform compare and swap.
                  If it fails, try again */
```

CDSG – Compare double and swap 16-byte values

This function is used to compare and swap 16-byte values in a storage location that is accessed by multiple I-streams.

Syntax:

Assembler: None

C:

int cdsg(cdsg_t *oldptr, cdsg_t *curptr, cdsg_t newvalue);

Oldptr	The address of a 16-byte value that is compared to the value pointed to by the **curptr** parameter.
Curptr	The address of a 16-byte value that is to be replaced by the **newvalue** parameter.
Newvalue	The value to be copied to the address specified by the **curptr** parameter.

Other variants in C: None

Entry Conditions: None

Return Conditions:

Assembler: None

C:

- The **newvalue** parameter value is copied to the storage location pointed to by the **curptr** parameter. A value of 0 is returned to the caller.
- A value of 1 is returned if the 16-byte values pointed to by the **oldptr** and **curptr** parameters are not equal. The value pointed to by the **curptr**parameter is copied into the storage location pointed to by the **oldptr** parameter.

Exception: None

Programming Consideration:

- The value pointed to by the **curptr** parameter must be on a 16-byte boundary.
- **Note:** Do not use a variable on the C language application stack for the current value; the stack might not start on a 16-byte boundary.

Example:

Assembler:

C:

The following example shows that 1 is added to the first doubleword in storage and 1 is subtracted from the second doubleword in storage. It also shows one method to ensure that the storage pointed to by the ***curptr*** *parameter is on a 16-byte boundary.*

```
    #include <tpf/cmpswp.h>
    #include <stdlib.h>
    cdsg_t *curptr;   /* pointer to current quadword value */
    cdsg_t  oldvalue; /* old quadword value                */
    cdsg_t  newvalue; /* new quadword value                */
    void * mallocptr; /* malloc storage address            */
    /* used to align storage on quadword boundary    */
    unsigned long int bdyalign;
    /* get storage                        */
    mallocptr = malloc(sizeof(cdsg_t) + 16);
    /* prepare to align on boundary       */
    bdyalign = (unsigned long int) mallocptr;
    /* align the storage address          */
    bdyalign = (bdyalign+15) >> 4;
    /* on a 16-byte boundary              */
    bdyalign = bdyalign << 4;
    /*use aligned addr for current ptr     */
    curptr = (cdsg_t *) bdyalign;
    /* initialize current value            */
    curptr->quad_word.first_dword = 100;
    /* initialize current value            */
    curptr->quad_word.second_dword = 100;
    /*set the old val to the current value */
    oldvalue = *curptr;
    do {
        oldvalue = *curptr;
                  /* Set the new value */
        newvalue.quad_word.first_dword=
                 oldvalue.quad_word.first_dword  + 1;
        newvalue.quad_word.second_dword=
                 oldvalue.quad_word.second_dword - 1;
      } while (cdsg(&oldvalue, curptr, newvalue) != 0);
    /* Perform compare and swap.  If it fails,
                       try again */
   free(mallocptr);
```

IPRSE_PARSE – Parse a text string against a grammar

IPRSE_parse matches an input string against an input grammar and produces a structure containing elements of the grammar with the corresponding elements of the input string. It is intended primarily for parsing z/TPF commands by real-time segments written in C language.

IPRSE_parse returns the parsed parameters and values through a pointer to a struct IPRSE_output, declared in the tpfparse.h header.

Syntax:

Assembler: None

C:

int IPRSE_parse(char *string, const char *grammar,

struct IPRSE_output *result, int options,

const char *errheader);

String	The input string, which must be a standard C string terminated by a zero byte ('\0') or by an EOM character if the IPRSE_EOM option is specified. If the IPRSE_EOM option is specified, the maximum length of the string is 4095 characters.
Grammar	The grammar describing acceptable input strings. The grammar must end in a zero byte ('\0').
Result	The tokenized parameter list in the following form: **`result.IPRSE_parameter`** The parameter name as specified by the grammar **`result.IPRSE_value`** The value of the parameter as specified by the input string. **Note:** This value is translated to upper case when the IPRSE_MIXED_CASE option is specified and the corresponding grammar parameter ends with a less-than sign (<). **`result.IPRSE_next`** The pointer to the next entry in the output parameter list.
Options	The following options control sending error messages: **`IPRSE_PRINT`** Print all error messages. **`IPRSE_NOPRINT`** Suppress all error messages. This is the default if IPRSE_PRINT is not specified. The following options control allocation of storage for the results:

IPRSE_ALLOC

Obtain storage dynamically for the output structure. Always code this option.

IPRSE_NOALLOC

This is the default if IPRSE_ALLOC is not specified. Do not use IPRSE_NOALLOC except to facilitate migration of old code that uses the`IPRSE_bldprstr` function to initialize preallocated storage. Using the IPRSE_ALLOC option is both more efficient and less likely to cause errors.

The following options are pertinent to input message blocks:

IPRSE_EOM

If the input string ends with an EOM character (+) instead of an EOS character ('\0'), the *IPRSE_parse* function replaces the EOM character with an EOS character. The first EOM character in the input string is replaced, there cannot be any '+' characters within the input string if the IPRSE_EOM option is specified. The EOM character must be in the first 4095 characters of the input string.

IPRSE_NOEOM

This is the default if IPRSE_EOM is not specified. The input string must end with an EOS character ('\0').

The following four options control how the `IPRSE_parse` function parses the input string. All four of these options can also be specified at the beginning of the grammar. Options specified in the grammar parameter override options specified in the options parameter:

IPRSE_STRICT

Accept only spaces as token separators, and only dashes (-) as separators between keywords and values in the input string. Use this option when the input parameters can contain commas (,), slashes (/), or equal signs (=).

IPRSE_NOSTRICT

Accept spaces, commas (,) or slashes (/) as token separators, and dashes (-) or equal signs (=) as separators between keywords and values. This is the default if IPRSE_STRICT is not specified.

IPRSE_MIXED_CASE

Accept lowercase or uppercase letters in the input string.

IPRSE_NOMIXED_CASE

Accept only uppercase letters in the input string. This is the default if IPRSE_MIXED_CASE is not specified.

IPRSE_QUOTE

Use the dollar sign ($) or single quote (') characters to delimit quoted parameters. The returned value will match the contents of a quoted parameter minus the delimiter. To specify a delimiter character, double up the character value; for example, 2 single quotes (' ') will return 1 single quote (').

IPRSE_NOQUOTE

This is the default if IPRSE_QUOTE is not specified. The IPRSE_

	NOQUOTE option specifies that there is no special meaning to the dollar sign ($) or single quote (') characters. Multiple options can be ORed together; for example, IPRSE_ALLOC | IPRSE_PRINT.
Errheader	A string that identifies the program calling the parser. This string is printed out as part of the error message text if the IPRSE_PRINT option is specified.

Other variants in C: None

Entry Conditions: None

Return Conditions:

Assembler: None

C:

- IPRSE_parse returns the number of parameters that have been parsed and put in the result structure. For example, a return code of 3 would mean that 3 parameters were parsed from an input string that contained 3 parameters.

- **Note:** The return code must be used to count the nodes when traversing the result.

- IPRSE_parse detects errors in the input string and in the grammar.
- Error in Input String
    - Return Codes
        - **-1**
    - The input string is a question mark (?) or HELP (represented by symbolic IPRSE_HELP).
        - **0**
    - The input string does not meet the requirements of the grammar (represented by symbolic IPRSE_BAD).
    - Error Messages
        - The IPRSE_parse function issues the following messages. The cccc represents the errheader parameter that is printed after the message header; it shows which function or program was calling IPRSE_parse when the error occurred. All messages will be sent via the wtopcfunction without chaining.

      PRSE0001E
    - cccc - TOO MANY PARAMETERS ENTERED

      PRSE0004E
    - cccc - INVALID USE OF PERIOD

      PRSE0005E
    - cccc - INVALID ALPHANUMERIC CHARACTER

      PRSE0006E
    - cccc - INVALID DECIMAL CHARACTER

PRSE0007E

- cccc - INVALID CHARACTER

PRSE0008E

- cccc - INVALID HEXADECIMAL CHARACTER

PRSE0009E

- cccc - MANDATORY PARAMETER NOT GIVEN
- If the system can determine the last parameter in error, the message indicates the last parameter in error by adding the PARAMETER IN ERROR IS text and the parameter value.

PRSE0011E

- cccc - INVALID INPUT PARAMETER
- If the system can determine the last valid parameter, the message indicates the last valid parameter by adding text that states LAST VALID PARAMETER IS and the parameter value. If a keyword was found to be in error, text will be added that states ERROR IN KEYWORD and the keyword.

PRSE0014E

- cccc - TOO MANY CHARACTERS ENTERED

PRSE0015E

- cccc - TOO FEW CHARACTERS ENTERED FOR PARAMETER

- Error in Grammar
  - 00006F system error messages are displayed in console or dump when the grammar syntax is in error.
- 0007B system error messages occur when the parser is unable to obtain needed heap storage.

Exception: None

Programming Consideration:

- Always code the IPRSE_ALLOC option. IPRSE_NOALLOC and the IPRSE_bldprstr are supported only for code that was written before the IPRSE_ALLOC option was available.

Example:

Assembler:

C:

This example shows a segment that parses a message in MI0MI format on data level D0. includes calls to:
- parse input using a specific grammar (IPRSE_parse)
- use the parsed output (process_parm, defined in this segment)

```
#include <tpf/tpfeq.h>
#include <tpf/tpfapi.h>
#include <string.h>
#include <stdlib.h>
#include <tpf/tpfparse.h>
```

```
/*Define the grammar for the command handled by
  this segment, where: Positional is a positional
  parameter. d+++ is a positional parameter that
  represents 1 to 4 digits.

  a.a is a positional parameter that represents
  a regular list of alphanumeric characters
 (character type a).

 (xx)* is an optional positional parameter that
  represents a wildcard list of hexadecimal digits
 (character type x).

 (NO)SELFdef is a self-defining keyword that returns
  a Y (yes value) or N (no value).

 Key-w is an optional regular keyword parameter
 that can have an alphanumeric value
 (character type w).

 List-cc.cc is an optional regular keyword parameter
 that can have a value that consists of a regular
 list of uppercase letters (character type c).*/

#define XMP_GRAMMAR "{ Positional "
                "| d+++ a.a [(xx)*] "
                "| (NO)SELFdef [Key-w List-cc.cc] "
                "}"
/*Declare an interface to functions that will
 process the parsed    command parameters.  */

enum parm1_type { POSITIONAL_NOT_SPECIFIED,
                  POSITIONAL_SPECIFIED };
enum parm5_type { SELFDEF_NOT_SPECIFIED, SELFDEF_NO,
                  SELFDEF_YES };
struct xmp_interface
{
  enum parm1_type   parm1_value;      "Positional"
  int               parm2_value;      "d+++"
  char             *parm3_first;      first "a"
  char             *parm3_second;     second "a"
  char             *parm4_string;     "(xx)*"
  enum parm5_type   parm5_value;      "(NO)SELFdef"
  char              parm6_value;      "Key-w"
  char             *parm7_first;      "first "cc"
  char             *parm7_second; second "cc"
};

#define XMP_DEFAULTS { POSITIONAL_NOT_SPECIFIED, -1,
                       NULL,NULL,NULL,
                       SELFDEF_NOT_SPECIFIED, '\0',
                       NULL, NULL }

/*Declare internal function called by this segment*/

static void process_parm(struct xmp_interface *xi ,
                         char *p, char *v);

/*Function completes the parsing of the "Zxxxx"
functional     message contained in the core block
on data level D0.  */

void _(void)
{

/*  Define variables for accessing the command
Text in the Core block on D0. */

struct mi0mi *block_ptr;  pointer to core block
char  *input_ptr;  pointer to message text
char  *eom_ptr; pointer to _EOM character

/*Define variables for the parser results.*/

struct IPRSE_output   parse_results;
int  num_parms;

/* For saving the IPRSE_parse  define
  a moving pointer for traversing the
  parse results, a wtopc header for the
  help message, and an interface variable
  for the parsed parameter values */

struct IPRSE_output  *pr_ptr;
struct wtopc_header   msg_header;
struct xmp_interface  parm_values = XMP_DEFAULTS;


/* Access the command block on level D0, point
  to the beginning of the parameters by
  skipping over "Zxxxx", and replace EOM
  with '\0'.*/

block_ptr = ecbptr()->ce1cr0;
input_ptr = block_ptr->mi0acc + strlen("Zxxxx");
eom_ptr =  (char *)&block_ptr->mi0ln0 + block_ptr->mi0cct-
1;
*eom_ptr = '\0';

/* Call the parser. */

num_parms = IPRSE_parse(input_ptr, XMP_GRAMMAR,
             &parse_results,
             IPRSE_ALLOC | IPRSE_PRINT, "cpp_tppc_test");

  /* Check if the command meets the
     grammar's requirements. */

if (num_parms > 0)

  /*The parse was successful; num_parms
    parameters from the command matched
    parameters specified in the grammar
    (XMP_GRAMMAR).*/

{
     pr_ptr = &parse_results;
                 /*  point to the first result    */
     do
     {
         process_parm(&parm_values,pr_ptr-
                       >IPRSE_parameter,
                       pr_ptr->IPRSE_value);
         pr_ptr = pr_ptr->IPRSE_next;
     } while (--num_parms);

  /* call additional functions to further
     process the input */

}
else
if (num_parms == IPRSE_HELP)
  {
     wtopc_insert_header(&msg_header,
                         "cpp_tppc_test", 99, 'I',
                         WTOPC_SYS_TIME);
     wtopc("EXAMPLE HELP MESSAGE", 0, WTOPC_NO_CHAIN,
            &msg_header);
    }
     else ;

  /*  IPRSE_parse has already written an
      error message. */

     exit(0);

  /* Command processing is completed.   */
}

  /* Function process_parm sets the appropriate
    Interface variable field to the value
    corresponding to the matched parameter.*/
    static void process_parm(struct xmp_interface *xi ,
                            char *p, char *v)
{

/* Define the value that strcmp returns when
    the two strings passed to it are equal*/


#define STRCMP_EQUAL 0

/* Define a local variable to point to the
  dot in a list.         */

char *dot_ptr;


/*  Determine which parameter was matched and
    set up the appropriate interface field(s)
    with the matching values. */

if (strcmp(p, "Positional") == STRCMP_EQUAL)
{
    xi->parm1_value = POSITIONAL_SPECIFIED;
}

else if (strcmp(p, "d+++") == STRCMP_EQUAL)
{
    xi->parm2_value = atoi(v);
}

else if (strcmp(p, "a.a") == STRCMP_EQUAL)
{

    dot_ptr = strchr(v, '.');


/*  Point to the dot separating
  the two list sub-parameters.   */

    *dot_ptr = '\0';

/*  Divide the list parameter into two sub-strings */

    xi->parm3_first = v;
    xi->parm3_second = dot_ptr + 1;
}

else if (strcmp(p, "xx") == STRCMP_EQUAL)
{
    xi->parm4_string = v;
}
else if (strcmp(p, "(NO)SELFdef") == STRCMP_EQUAL)
{
    xi->parm5_value =
        *v == 'Y' ? SELFDEF_YES : SELFDEF_NO;
}

else if (strcmp(p, "Key-w") == STRCMP_EQUAL)
{
    xi->parm6_value = *v;
}

else if (strcmp(p, "List-cc.cc") == STRCMP_EQUAL)
{
    dot_ptr = strchr(v, '.');
    *dot_ptr = '\0';
    xi->parm7_first = v;
    xi->parm7_second = dot_ptr + 1;
}

return;
}
```

12. zTPF – Programming Conventions

12.1 Program Name

- The zTPF program names are 4-character in length and suffixed by a 2-character version number.

- Both program name and version are alphanumeric. Multiple versions of the same program can be loaded in the system. When a new version is loaded in the system, an existing ECB will use the old version it was referring till the exit.

12.2 BEGIN and FINIS macro

- All E-type program must begin with a BEGIN macro and end with a FINIS macro.

- The BEGIN Macro:

  - Formats the program header
  - Defines the CSECT statement
  - We can specify multiple registers in BASE option to base code beyond 4K BASE = (R_X, R_Y) Rx base for first 4K and R_Y base for second 4K (valid register 1-8).
  - Calls the equate macros such as UXTEQ, SYSEQ, EB0EB

- The FINIS Macro:

  - Marks the date and time of assembly
  - Calls the equate macros CPSEQ and REGEQ

12.3 Program Structure

In general, a zTPF program has the following structure:

BEGIN statement

Program documentation

Revision header

DSECT statements

Macro declarations

Code section – Main-line

Code section – Subroutines

Constant definition

Literal section

FINIS statement

13. Frequently used zTPF Commands

ZDFIL – Display File

Syntax	ZDFIL <*file address*> <*displacement*>.<*length*>
Description	To display the contents of a file.
Parameters	*file address* F.A. of the file to be displayed. *displacement* Displacement from where the display must start. *Length* Number of bytes (in hexadecimal) to be displayed.
Instance	ZDFIL 758036C9 000.4F DFIL0011I 03.44.57 BEGIN DISPLAY - FILE ADDRESS 00000000758036C9 00000000- C1C1A1A0 C3E5C1C1 00000000 00000000 AA~.CVAA .... 00000020- F880F0F2 D7D90000 02000000 00000000 8.02PR.. .... 00000030- 000003A1 41040401 FAE7C4C2 D2D9B000 ...~.... .XDBKR.. 00000040- 00000000 00000000 00002310 00000000 ....

ZAFIL – Alter File

Syntax	ZAFIL <file address> <displacement> <data>
Description	To alter the contents of a file.
Parameters	file address F.A. of the file to be altered. displacement Displacement to be altered. Data Contents to be replaced at the specified displacement.
Instance	ZAFIL 758036C9 039 C4C5D5 AFIL0011I 03.53.37 BEGIN DISPLAY - FILE ADDRESS 00000000758036C9 00000038- FAE7C4C2 D2D9B000 00000000 00000000 .XDBKR.. .... 00000048- 00002310 00000000 00010000 016DB1C9 ...._.I ALTERED TO BY LNIATA 0401FA 00000038- FAC4C5D5 D2D9B000 00000000 00000000 .DENKR.. .... 00000048- 00002310 00000000 00010000 016DB1C9 ...._.I END OF DISPLAY - ZEROED LINES NOT DISPLAYED

ZDDAT – Display Date

Syntax	ZDDAT
Description	To display the system date.
Instance	ZDDAT DDAT0001I 04.01.45 SUBSYSTEM BSS DATE IS 29JUL11

ZDTIM – Display Time

Syntax	ZDTIM
Description	To display the current system time.
Instance	ZDTIM DTIM0001I 04.03.04 SUBSYSTEM BSS LOCAL STANDARD TIME

ZDSYS – Display System Operating State

Syntax	ZDSYS
Description	To display the current system operating state, which can be one of the following: • 1052 State • UTIL state • CRAS state • Message Switching state • NORM state
Instance	ZDSYS DSYS0001I 04.05.02 THE SYSTEM IS IN NORM STATE

ZDECB – Display In-Use ECBs

Syntax	ZDECB OLD <n | ALL> ZDECB STAT <n | ALL>
Description	To display information about ECBs that are in-use.
Parameters	OLD Oldest n or ALL ECBs in the system. STAT Statistics of last n or ALL ECBs in the system.
Instance	See below

```
ZDECB OLD ALL

DECB0014I 04.09.03 DISPLAY ECB SUMMARY
ECB ADDR  SSU IS  PGM   MIN SC  ORIGIN   I H DSP  SVC
057D0000 UASU  2 CDBI42   5  1   CXFR    1   DLM SAWNC
05968000 UASU  1 CVXS49      0             DLM CLHSC
TOTAL       2
END OF DISPLAY


ZDECB STAT ALL

DECB0014I 04.12.32 DISPLAY ECB SUMMARY
ECB ADDR IS  PGM   MIN SC    MILS  FRM   FIND   FILE   GETF
057D0000  2 CDBI42   8 29       29   52     10      0      0
057A6000  1 CTS248      1        0    9      0      0      0
05977000  1 CVXS49      0        0   30      2      1      0
TOTAL       3
END OF DISPLAY
```

ZDPGM – Display Program

Syntax	ZDPGM <*program name*> ZDPGM <*program name*> <*displacement*>.<*length*> *INSTR*
Description	To display a specified number of bytes from a program beginning at a specified displacement.
Parameters	*program name* 4-character program name. *displacement* Displacement within the program. *Length* Number of bytes (in hexadecimal) to be displayed. *INSTR* Option to display in disassembled format.
Instance	see below

Instance

```
ZDPGM UII1

DPGM0010I 04.16.17 BEGIN DISPLAY OF FILE COPY FOR
             UII1.G3 ACTIVE IN LOADSET BASE
00000000- 00FF0DAC E4C9C9F1 BF1E9349 4310932B ....UII1 ..l...l.
00000010- 41E08020 56E0801C 0B0E0000 80000000 ........ ........
END OF DISPLAY - ZEROED LINES NOT DISPLAYED

ZDPGM UII1 010.20 INSTR

DPGM0010I 04.15.41 BEGIN DISPLAY OF FILE COPY FOR
             UII1.G3 ACTIVE IN LOADSET BASE
00000010  41E0 8020      LA     R14,32(,R8)
00000014  56E0 801C      O      R14,28(,R8)
00000018  0B0E          BSM    R0,R14
0000001A  0000          ???
0000001C  8000 0000      SSM    0
00000020  58E0 B180      L      R14,384(,R11)
00000024  58E0 E3D8      L      R14,984(,R14)
00000028  58F0 E008      L      R15,8(,R14)
0000002C  0DEF          BASR   R14,R15
0000002E  1211          LTR    R1,R1
END OF DISPLAY
```

ZDMAP – Display C Load Module Link Map

Syntax	ZDMAP <*program name*>
Description	To display link map data of a C load module. The Link map data contains the names and addresses of all the object files and C functions within a load module.
Parameters	*program name* 4-character program name of the C load module.

Instance

```
ZDMAP YFU0

DMAP0001I 04.25.40 LINK MAP DATA DISPLAY
             YFU0G2 ACTIVE IN LOADSET BASE
             C LOAD MODULE ADDRESS - 14928020
             C LOAD MODULE SIZE    - 0000326C

YFU0A2   IS AN OBJECT FILE AT ADDRESS      14928078
  OBJECT FILE SIZE - 00002170
  COMPILED ON 2004/09/22 AT 08.27.38

END OF DISPLAY
```

ZDPAT – Display Program Allocation Table

Syntax	ZDPAT <*program name*>
Description	To display the values assigned to the allocation parameters of a program. The following information is displayed: • Program version code • Program type • File address of the program • Addressing mode in which the program was entered (24-bit or 31-bit) • Macros that the program is authorized to use
Parameters	*program name* 4-character program name.
Instance	ZDPAT UII1 BEGIN DISPLAY OF CORE COPY PROGRAM UII1 VERSION G3 BASE PAT SLOT 14519C00 TYPE CORE RESIDENT PRELOAD LINKAGE TYPE BAL CLASS COMMON FILE ADDRESS 794158FD ADDRESSING MODE 24BIT AUTHORIZATION KEY0 MONTC RESTRICT TEST HOOK NONE ADATA FILE ADDR ADATA NOT LOADED END OF DISPLAY

ZOLDR – Load/Activate/Deactivate/Delete a Loadset

Syntax	ZOLDR <*LOAD* | *ACT* | *DEA* | *DEL* | *DIS L*> <*lsname*>
Description	To load, activate, deactivate and delete loadset.
Parameters	*lsname* 5-8 character alphanumeric loadset name
Instance	ZOLDR LOAD TESTLB ZOLDR ACT TESTLB ZOLDR DEA TESTLB ZOLDR DEL TESTLB ZOLDR DI L TESTLB