TPF DECB Support

Michele Dalbo, IBM TPF ID Core Team, and Jason Keenaghan and Mike Wang, IBM TPF Development

What Is a DECB?

A data event control block (DECB) can logically be thought of as another data level in an entry control block (ECB). A DECB can be used in place of an ECB data level for FIND/FILE-type I/O requests by applications. Although a DECB does not physically reside in an ECB, it contains the same information as standard ECB data levels: a core block reference word (CBRW), a file address reference word (FARW), file address extension words (FAXWs), and a detailed error indicator (SUD).

Before TPF DECB support, the TPF 4.1 system restricted the number of ECB data levels (D0–DF) that were available for use to 16 (the number of data levels defined in the ECB). DECBs can be acquired dynamically by a single ECB (through the use of the tpf_decb_create C++ function or the DECBC assembler macro), with no preset limit on the number of DECBs that a single ECB can create.

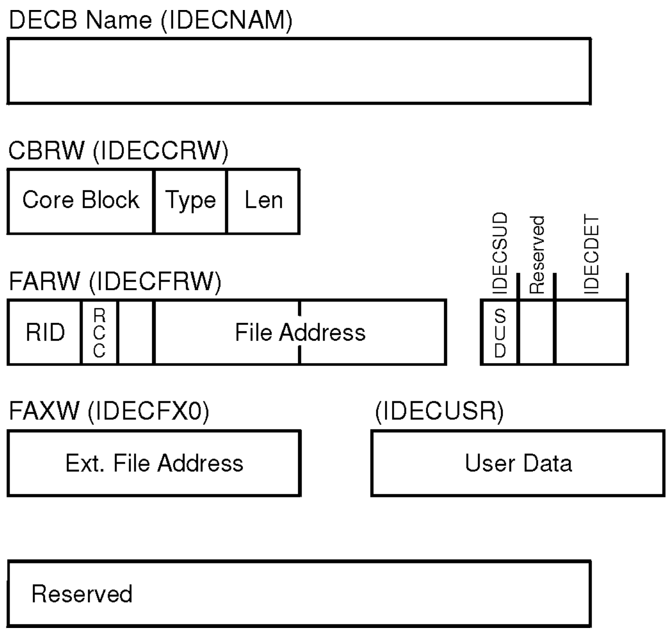
How Is a DECB Different from an ECB Data Level?

An ECB data level is simply a series of doubleword reference or control fields (CBRWs, FARWs, and FAXWs) for data blocks that can be used however the application requirements dictate. A DECB is basically the same except the file address field in the FARW has been expanded to allow for 8-byte file addresses as compared to 4 bytes in an ECB data level FARW. Because there is no room for that expansion in an ECB FARW without changing the entire layout of the ECB, a DECB is the logical place for 8-byte file addresses to be stored.

In addition, using TPF DECB support instead of ECB data levels allows you to associate symbolic names with each DECB. This feature allows different components of a program to easily pass information in core blocks attached to a DECB. Each component only needs to know the name of the DECB where the information is located to access it.

What Are the DECB Fields?

The following figure shows the DECB application area and the fields inside the DECB.



The DECB fields are used as follows:

IDECNAM	Contains the name of the DECB.
IDECCRW	Corresponds to an ECB core block level.

IDECDAD (Core block) - Contains the address of a core block.
IDECCT0 (Type) - Contains the core block type indicator or X'0001' to indicate there is no core block attached.
IDECDLH (Len) - Contains the data length.

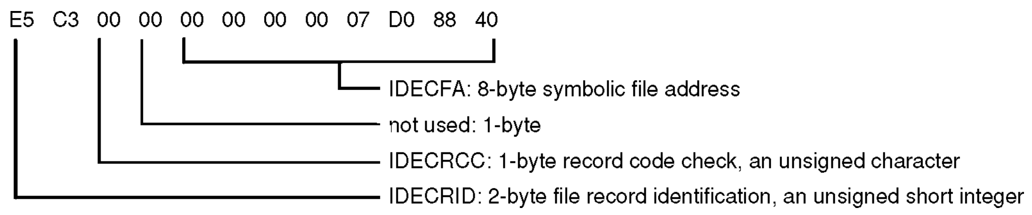
IDECFRW Corresponds to an ECB FARW.

IDECRID (RID) - Contains the record ID for a FIND/FILE request.
IDECRCC (RCC) - Contains the record code check (RCC) value for a FIND/FILE request.
IDECFA (File Address) - Contains the file address for a FIND/FILE request.

IDECSUD	When the FIND/FILE request is completed, this field will be set to the SUD error value, or zero if there is no error.
IDECDET	Contains the number of core blocks currently detached from this DECB.
IDECFX0	Contains the FAXW information.
IDECUSR	Contains the user data.

What Is 4x4 Format?

The IDECFA field, which contains the file address, is 8 bytes in the DECB. This allows 8-byte file addressing in 4x4 format, which provides for standard 4-byte file addresses (FARF3, FARF4, and FARF5) to be stored in an 8-byte field. This is done by having the file address reside in the low-order 4 bytes of the field. The high-order 4 bytes of the field contain an indicator (a fullword of zeros) that classifies it as a valid 4x4 format address. When there is file activity, the FARW will be used for the record identification (2 bytes), record code check (1 byte), and the symbolic file address (8 bytes); 1 byte is unused. The following example shows the layout of a 4x4 format file address:


E5 C3 00 00 00 00 00 00 07 D0 88 40
IDECFA: 8-byte symbolic file address
not used: 1-byte
IDECRCC: 1-byte record code check, an unsigned character
IDECRID: 2-byte file record identification, an unsigned short integer

How Do I Create and Release DECBs?

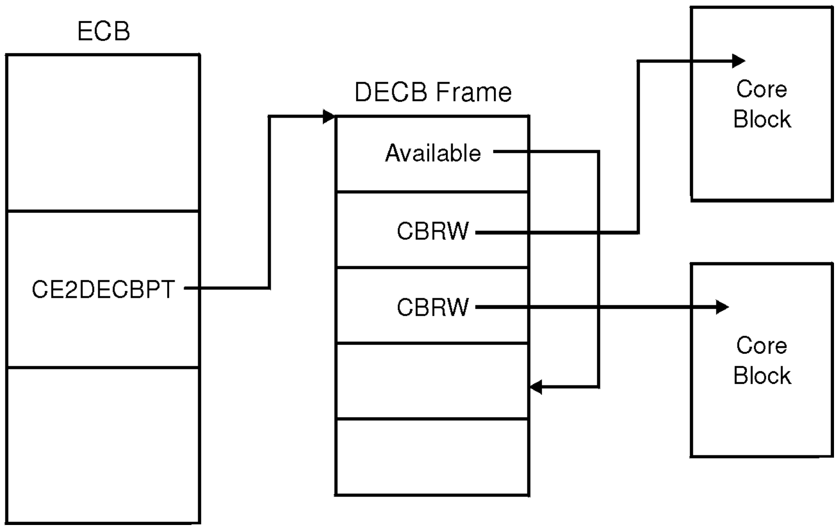
TPF programs can dynamically create and release DECBs by using the tpf_decb_create and tpf_decb_release C++ functions or the DECBC assembler macro. The storage, which will hold the DECB comes from the 1-MB ECB private area (EPA). Therefore, the number of DECBs that the ECB is restricted to is limited only by the amount of storage in the private area that is dedicated to DECBs.

When the first DECB creation request is received by the TPF system, a 4-K frame is obtained from the EPA. This frame, which will act as a DECB frame, is attached to page 2 of the ECB at field CE2DECBPT. The DECB frame itself contains a header section to help manage the DECBs. The remainder of the frame is then carved into the individual DECBs themselves; therefore, a single DECB frame contains several individual DECBs (currently a single frame can hold 31 DECBs). If more DECB creation requests are received than can fit in a single DECB frame, another 4-K frame is obtained and forward chained to the previous DECB frame.

Individual DECBs are dispensed sequentially using a first-in-first-out (FIFO) approach. The address of the next DECB to be dispensed is maintained in the header of the DECB frame. When dispensing the DECB, the contents are first cleared and the DECB is marked as not holding a core block. The DECB is then marked as in-use (that is, *allocated*) and the name of the DECB, if one was supplied at creation time, is assigned. Finally, the DECB frame header is updated to indicate the address of the next DECB that will be dispensed.

When a DECB is released by the application, the DECB is simply marked as not in use. The DECB is not cleared until it is later redistributed, nor is the DECB frame header updated to indicate the address of the next DECB to be dispensed. Once released, this DECB will not be redistributed until all other DECBs in this frame have been dispensed at least one time (FIFO approach). No DECB frames will be detached from the ECB until ECB exit time, even if all DECBs are currently marked as not in use. This is done to speed processing if another DECB creation request is received by the TPF system.

The following figure shows the relationship between DECBs and the ECB.



How Do I Access File Records with a DECB?
All types of applications can use DECBs. Application programming interfaces (APIs) have been added or updated to allow TPF programs to access file records with a DECB instead of an ECB data level. However, only a subset of the existing macros and C functions that currently reference ECB data levels accept a DECB in place of an ECB data level. Macros and C functions that use a file address will first verify that it is a valid address in 4x4 format. If the address is not valid, a system error will occur.

How Does a DECB Handle Errors?
With TPF DECB support, each DECB has a detailed error indicator byte (IDECSUD). The existing CE1SUG byte will include any errors that occur on a DECB-related I/O operation. In addition, the IN1DUD bit in the CE1IN1 byte of the ECB will be set to indicate that an error occurred on a DECB.

What Other Areas of TPF Will Be Affected by the Introduction of DECBs?
The following areas in the TPF system have been changed to best make use of TPF DECB support:

- The ZFECB functional message has been modified to display DECBs attached to a given ECB.

- The real-time trace (RTT) utility has been modified to save and display information about DECBs for specific types of macros that are called by a particular ECB.

- ECB macro trace has been modified to handle DECBs as macro parameters instead of ECB data levels as well as to correctly handle 8-byte file addresses.

- Because of the increased size of each macro trace entry, the number of macro trace entries for each ECB has been reduced from 66 to 55 when tracing without registers, and from 25 to 23 when tracing with registers.

- System error processing has been updated to gather information about DECBs attached to an ECB when an error occurs. Additionally, system error processing now searches for any core blocks attached to or detached from DECBs; this will also be reported in a system dump.

- The interface to the Dump Data user exit (segment CPSU) has been updated to pass along information about DECBs to the user if an error occurs.

- ECB exit processing has been modified to release any memory resources associated with the DECB before stopping an ECB.

Why Must I Use the C++ Compiler Instead of the C Compiler When Using DECBs?
When creating the new C language functions that would support DECBs and, in some cases, 8-byte file addresses without DECBs, several different approaches were examined. The primary goal was to create functions that could be quickly learned by experienced TPF programmers. In other words, if a new C function was going to be created to release a core block on a DECB, it should look very similar to the existing function to release a core block on an ECB data level (`relcc`).

Note: In the remainder of this article, `relcc` is used as an example of the functions updated for TPF DECB support.

The first approach would have been to change the existing function to accept a pointer to a DECB (`TPF_DECB *`) in addition to an ECB data level (`enum t_lvl`). However, this would have forced all existing applications to be recoded and recompiled to account for the additional parameter that was added. This is obviously an unacceptable solution.

A second approach would have been to create completely new functions with new names and new parameters to handle similar functions with DECBs instead of ECB data levels. This approach becomes cumbersome to a TPF application programmer who would have to recognize that when using ECB data levels, the correct function to call would be `relcc`; however, when using DECBs, the correct function to call would be something like `tpf_relcc_decb` (to follow TPF naming conventions).

The final approach, and the one that has been implemented for TPF DECB support, is to create functions with the same name (relcc), but have the function take a pointer to a DECB instead of an ECB data level as a parameter. The benefits of this approach include the following:

- Eliminating the need to recompile any existing applications that call the function
- Using familiar TPF function names so application programmers do not have to remember additional functions.

To enable existing TPF C functions to use DECBs instead of ECB data levels, the functions have been ***doubly defined*** by using the C++ function overload feature. For example, in addition to the existing `relcc` function prototype:

```
#ifdef __cplusplus
  extern "C"
#endif
void relcc (enum t_lvl level);
```

an overloaded relcc function that takes a DECB pointer as a parameter has been created:

```
#ifdef __cplusplus
  } /* end extern "C" */

      void relcc (TPF_DECB *decb);
#endif
```

The overloaded function uses C++ function linkage, so programs that use overloaded functions must be compiled with the C++ compiler.

In TPF, the `relcc` function (which uses ECB data levels) is actually an assembler program (crelcc.asm) that resides in the CTAL (TPF Application Library) ISO-C library. This did not change with the introduction of the overloaded `relcc` function (which uses DECBs). Any calls made to `relcc` using ECB data levels will still go to the CTAL library; it does not matter if the call is made from a C or C++ program.

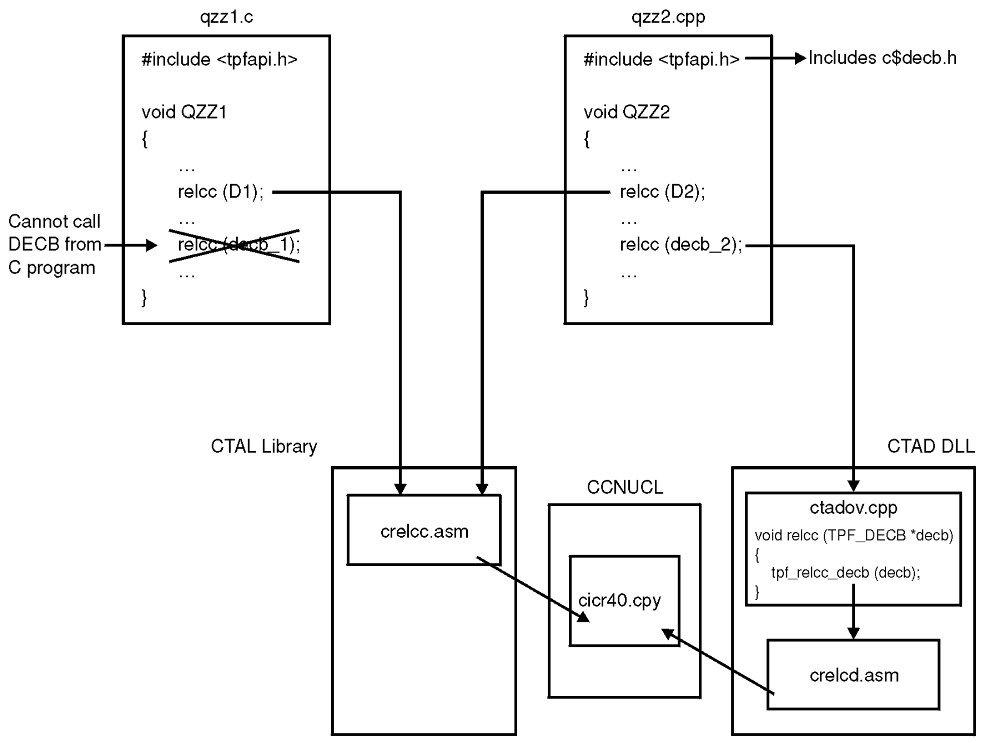
The definition of the `relcc` function, which uses DECBs, is part of a C++ segment that contains what can be thought of as *bridge* functions (segment ctadov.cpp). This segment has been included as part of the new CTAD (TPF Application DLL) DLL. The underlying service routine (or *stub*, as it is sometimes called) is an assembler language program. However, C++ and assembler programs cannot communicate between each other directly, but C++ programs can communicate with C programs and C programs can communicate with assembler language programs. Therefore, the new bridge functions have been created to span the gap between C++ and assembler programs. The following shows an example from segment ctadov.cpp:

```
#pragma export(relcc(TPF_DECB *decb)) <— This makes relcc callable by other DLMs and DLLs
.
.
#pragma map(tpf_relcc_decb, "@@RELCCD")
extern "C" tpf_relcc_decb (TPF_DECB *decb); <— This is internal to this DLL and generates C-type
                                               linkage
.
void relcc (TPF_DECB *decb)
{
        tpf_relcc_decb (decb);  <— Call internal C function to process relcc request
}
```

The functions shown in the previous example interact in the following way: a pseudo C function called tpf_relcc_decb has been created. The underlying service routine for this C function can then be written in assembler language, much like the service routine for relcc (enum t_lvl level) is in the CTAL library. To avoid recompiling or reassembling, a separate segment was added to process the DECB parameter. The segment that processes tpf_relcc_decb is in an assembler segment (crelcd.asm), which is also a member of the CTAD DLL. The first few instructions in segment crelcd.asm would look as follows to be correctly resolved at when the DLL is linked:

```
              BEGIN NAME=CRELCD,IBM=YES,VERSION=40,TPFISOC=YES
              ...
@@RELCCD      TMSPC FRAMESIZE=NO,LWS=R6,PPANAME=tpf_relcc_decb
              L       R2,0(R1)                GET DECB FOR RELEASE
              ...
```

When a TPF application wants to call the `relcc` function using DECBs, the TPF application must first be compiled with the C++ compiler. This does not mean that the application has to be written as an object-oriented C++ application; it only means that the `__cplusplus` declaration must be defined (which happens automatically when the segment is a .cpp file). The second step is to ensure that the definition side-deck for the CTAD DLL is imported correctly when linking the dynamic load module (DLM) containing the application calling the overloaded function. This enables the correct function linkage to be generated to the relcc service routine in the CTAD DLL at run time. The following shows an example of how all this links together for both C and C++ programs. Notice that both functions, in the end, call the same SVC service routine in the CCNUCL CSECT of the control program (CP):



Where Can I Find More Information about TPF DECB Support?

For more information, see the following books:

- *TPF Application Programming*
- *TPF C/C++ Language Support User's Guide*
- *TPF Concepts and Structures*
- *TPF Database Reference*
- *TPF General Macros*
- *TPF Library Guide*
- *TPF Messages*
- *TPF Migration Guide*
- *TPF Operations*
- *TPF Program Development Support Reference*
- *TPF System Generation*
- *TPF System Installation Support Reference*
- *TPF System Macros*

INTERESTED IN TPF CERTIFICATION?

If you teach, train, educate, tutor, school, or otherwise instruct people in TPF technical matters, you are welcome to join us at the next TPF User Group Conference in Toronto, Canada April 23-25, 2001

... as we shape the future of TPF Education.

Parity Expands Presence in the TPF Euromarket

Parity EuroSoft, the European IT resourcing and recruitment division of Parity Group plc, has announced the formation of a dedicated new TPF business unit designed to meet the growing demands of clients who require staffing solutions in this sector.

Used predominantly by major organisations in the airline, banking, credit card and hotel industries, TPF technology is now key to many systems that rely on the ability to handle a large volume of transactions at any one time.

Parity Eurosoft's TPF business unit will be led by Dave Hayton who has worked in the TPF IT recruitment industry for over 16 years and has been with Parity since 1995. He brings technical and practical experience to the role having previously been involved in over 30 TPF contracts for European based airline.

Dave said: "Parity has decided to expand its presence in TPF because of the growing need of our customers in this field. With only 9,000 consultants worldwide, TPF skills are much sought after and it is very much a niche market. Parity's global network will be an important factor in the unit's success as it gives us access to qualified consultants worldwide. I have already recruited consultants from South America, South East Asia, India, and Australia as well as from the UK and Europe.

Parity Eurosoft's sister company, Parity TelTech, is already working in TPF in the United States and has been very successful in delivering to this sector of the systems market. Our focus will be the European market where we will be using Parity's global reach to deliver TPF solutions to our customers."

Parity's Software Services division, part of Parity Group plc, a TechMARK 100 company, comprises Parity Resources, Parity Selection, Parity EurosSoft and Parity TelTech. The division, which provides integrated IT resourcing and recruitment solutions throughout the UK, mainland Europe and the USA, recently announced turnover of 103.7 million Pounds in the six months to 30 June 2000.

For further information contact: Julie Mazzei - IMA Public Relations, Park House, Didsbury Park, Didsbury, M20-5LT - Tel: 0161 434 9125; Fax: 0161 434 9427; Email: julie@i-m-a.co.uk