```
CP_ddclose(fp);
```

The Checkpoint I/O API For Record-Oriented Files

The checkpoint library supports its own I/O API for record-oriented sequential files. This API was originally intended to support COBOL development, but is also callable from C or any other language.

The record-oriented API uses a data structure known as the I/O Control Area (abbreviated IOCA) to communicate information about a record-oriented file between an application program and the checkpoint API. This data structure is similar to the C `FILE` structure, or an FD in COBOL. Each I/O API call takes the IOCA as the first parameter. Each file which is opened using the record-oriented I/O API must have its own unique IOCA, since the checkpoint library uses fields in the IOCA to store information about files which speeds up the performance of I/O calls.

MAINFRAME MIGRATION TIP

Warning: The I/O Control Area structure is not compatible between the mainframe version of the checkpoint library and the non-mainframe version.

The following is the definition of the I/O Control structure for (32-bit) COBOL:

```
01 IO-CONTROL-AREA
         05    IO-DDNAME          PIC X(08).
         05    IO-STATUS-CODE     PIC X(02).
         05    FILLER             PIC X(06).
         05    IO-TYPE            PIC X(04) VALUE SPACES.
         05    IO-R-OR-L          PIC  X(1).
         05    IO-F-OR-V          PIC  X(1).
         05    IO-FS              PIC  X(1) VALUE SPACE.
         05    FILLER             PIC  X(1).
         05    IO-MAXREC          PIC  9(9) COMP VALUE ZERO.
         05    IO-MINREC          PIC  9(9) COMP VALUE ZERO.
         05    IO-FI              PIC  9(9) COMP VALUE ZERO.
```

For 64-bit COBOL, note that the `PIC 9(9)` fields are machine words (long integers), and must be coded as `PIC 9(19)` on 64-bit platforms. (See the 64-bit section of this manual for more information on differences for 64-bit programs.)

The initializes `VALUE SPACES` and `VALUE ZERO` should *always* appear in the declaration. All data pictures must be exactly as shown. If the format shown here is not used, undefined behavior will result since the values, data type sizes, and offsets from the start of the structure of these fields are used by the checkpoint library.

In C, this structure is defined in the dbckpt.h header file.

```
struct io_tag {
         char      dd[8];
         char      status[6];
         char      filler[6]
         char      type[4];
         char      RorL;
         char      ForV;
         char      fs;
         char      filler2[1];
         long      maxrec;
         long      minrec;
         long      fi;
};
```

```
typedef struct io_tag io_control;
```

You create a control structure in C the same way you do any variable:

```
io_control io_c;
```

The structure must be explicitly initialized as described above in the COBOL section.

Following is an explanation of the fields in the I/O control area:

dd: this is the dd name of the file.

status: The status field is how the checkpoint API passes information back to the application.

Note: The STATUS field of the I/O Control Area data structure has codes which mirror the FILE STATUS codes in ANS COBOL. The four codes used by the I/O Control Area have the same basic meaning as the ones in COBOL: In COBOL, 00 means success, 10 means end of file, 41 means opening a file that is already open, and 42 means closing a file that has never been opened.

MAINFRAME MIGRATION TIP

On the mainframe, the status codes are formatted in standard mainframe unpacked number format, that is with the high-order half-byte in each character set to 'F'. (E.g.: 0xf4f2) This number format does not exist on non-mainframe platforms, so each of the two bytes of the status code is a regular 8-bit character (in the native machine collating sequence, which is normally ASCII).

This table shows the meanings of the status codes which the checkpoint library returns to application programs:

Status Code	*Means*
00	Successful completion of I/O
10	End of file
41	Open attempted on already open file
42	Close attempted on unopened file

type: This field determines how the file will be used. It can have the values READ, or WRTE for the OPEN call (and FEOV for the CLSE call, which is ignored under UNIX®).

MAINFRAME MIGRATION TIP

On the mainframe, file information coded in a job's JCL supplies many of the parameters which are explicitly included in the UNIX and Windows library's I/O Control Area. These additional fields in the I/O Control Area have been added to supplement the information in the mainframe library's I/O Control Area.

RorL: This byte is either coded as 'R' or 'L' for record sequential or line sequential. More information on these file types appears later in this manual.

ForV: The next byte is either 'F' or 'V', for fixed length records or variable length records.

maxrec: For variable length sequential files only, the IOCA must specify the maximum record length here. Otherwise, this value is not used.

minrec: For variable length sequential files only, the IOCA must specify the minimum record length here. Otherwise, this value is not used.

fs: an internal flag that the checkpoint library uses to determine the status of the file. This should always be initialized to a space. After initialization, it should be left alone.

fi: an internal index the checkpoint library uses. This should always be initialized to a space. After initialization, it should be left alone.

MAINFRAME MIGRATION TIP

Unlike applications running under mainframe JCL, where file organization can be strenuously checked against the actual file organization on disk, in UNIX and Windows there is no inherent file organization in data files against which to check the parameters in the I/O Control Area for accuracy. The stream files in UNIX and Windows have no inherent organization. All record formats are imposed on a file by the application itself, not the operating system. This means there is no way for the checkpoint library to validate the information about record lengths, etc in the I/O Control Area against the actual organization of files on disk. Attention needs to be given to the correctness of the numbers passed in the I/O Control Area.

Supported File Types For Record-Oriented Files

The checkpoint API supports three basic record formats: fixed-length sequential, variable-length sequential, and line sequential. Files created with the checkpoint API are designed to be compatible with MicroFocus COBOL, and have been tested successfully with COBOL programs. The three main file types are designed to mirror the sequential files in MicroFocus COBOL.

MAINFRAME MIGRATION TIP

The I/O Control Area has been extended in the non-mainframe version with fields which contain file characteristics normally found in mainframe JCL, such as record length, which have no analogue in the stream-oriented world of UNIX and Windows.

Fixed-length records are written sequentially, with no header or other markers in the file. Line sequential files are written with variable-length records delimited by specific platforms' end-of-line character (or characters, in Windows). Variable-length sequential files are written in a special MicroFocus COBOL format which has both a file header and record headers.

Line sequential support needs to be enabled by the LS option in the options file. By default, line sequential support is not enabled if it is not used, to reduce program overhead. The LS option's value is the maximum line sequential record length any record in the program would need. (LS's value controls an internal buffer the checkpoint API uses.) In order to open a file as line sequential, set the RorL field in the IOCA to "L". The other fields, ForV, maxrec, and minrec, are not used.

When a line sequential record is read, the buffer passed to the read call must be large enough to hold the largest expected record, and the length parameter to the READ call must be set to the maximum length for the record. The checkpoint API will read a record up to the number of bytes passed in the length parameter of the READ call, but may read fewer if the record delimiter is encountered. If a record is smaller than the buffer, it is space-padded to the end. (In other words, although the records in the file are variable-length, COBOL buffers for records are not.)

If the checkpoint API encounters a record in a line sequential file longer than the buffer length passed to the READ call, it behaves as the operating system does: it reads up to the maximum record length from the record, and reads the rest of the record on the next READ call. If the record's buffer is 100 bytes, and a 200 byte record is encountered, the call to read returns the first 100 bytes, and the next call returns the next 100 bytes.

End of file is different in line sequential files than it is in fixed-length record sequential files. A fixed-length sequential READ returns the last record in the file, and then returns EOF on the *next READ* call. It is impossible to read a partial record in a fixed-length sequential file, so EOF is always returned on the *next* attempted READ call, which returns EOF without reading a record. Line sequential files are the opposite: it is possible to read a record and encounter EOF at the same time, so the application must check for EOF after every line sequential read.

When the checkpoint API writes a line sequential record, it writes at most the number of bytes in the WRTE call's length parameter. If the record is shorter than the maximum length, the end must be space-padded. The checkpoint API goes *backwards* through the buffer from the end to the beginning, looking for the first non-space character, and assumes that is the end of the line. It writes that record along with the record delimiter for the platform.

The variable-length sequential file format is compatible with MicroFocus COBOL, and uses their file header and record header layout.

Some points of note about the record sequential variable length files:

Reserved fields: The checkpoint API sets all the reserve fields and unused fields in the file header to binary zeroes. MicroFocus-generated reserved fields occasionally have something in them, but the zeroes do not seem to inhibit MicroFocus COBOL's ability to read the files.

Date in header: The checkpoint API implements the MicroFocus COBOL record sequential variable length file format as closely to published specifications as possible. One area of ambiguity occurs in the file header of RSV files. In these files, two date fields are present. The published documentation says that these are used only for *indexed* files, which the checkpoint library does not support, and are zeroed out for sequential files. Dumps of MicroFocus-produced *sequential* files show these date fields to be present also. By default, the checkpoint library places zeroes in this reserved field, but the MF_DUPLICATE_DATE option can be used to place a date in them if necessary.

Maximum record length: The maximum record length is very critical. If it is up to 4095 bytes, variable length files have a *two byte* record header. If the maximum size is greater than 4095 bytes, the record headers are *four bytes*. Failure to specify a correct maximum record length will result in undefined behavior. Besides checking the length field to see if it is greater or less than 4095 bytes, COBOL does not seem to use this length, so the maximum length is mainly meaningful in setting the record format.

Maximum maximum record length: How big can the maximum record length be? Although a four byte header is used, not all of it is given to the size of the record. Four bits are used to determine what kind of record it is and the rest is given to the size. This means that the maximum record size is 2^{28}, or 268,435,456 bytes, not 2^{32} as a 32-bit integer would be.

Minimum record length: MicroFocus uses the minimum record length as a validity check. If the minimum record length in the COBOL application's FD for a file does not match what is in the header, the COBOL runtime system abends. Thus it is vitally important to pass the checkpoint API the correct minimum record length. We have not been able to determine any other use for the minimum record length. The checkpoint API performs no tests or checking on it. (*Warning*: The checkpoint API will happily ignore the minimum record length. It will read files where the length in the file does not match what you passed in the I/O control area, as long as everything else is valid and uses a legal file layout. If you intend your files to be mixed between record-oriented I/O with the checkpoint API and COBOL, the application must handle the minimum record length. If application program logic requires the minimum record length to be used for some reason, application must have tests against it.)

OPEN: Opening A Record-Oriented File

The OPEN call opens a file for either input or output. The type member of the IOCA should either be READ or WRTE to determine how the file is opened.

Example calls:

```
/* C */                              * COBOL
memcpy(io_c.type, "READ", 4);         MOVE "READ" TO IO-TYPE
dbckpt("OPEN", &io_c);                CALL CP-DBCKPT USING
                                            CP-OPEN, io-control-area
```

After the OPEN call returns, applications should check the return code, which is in the STATUS field of the IOCA.

READ: Reading A Record-Oriented File

The READ call will read the next record from a file. The length parameter is the length of the record (or maximum length of a variable-length record), and the buffer must be big enough to hold the record. The length parameter is an input parameter with the record length for fixed-length sequential files, and is an output parameter which the checkpoint API sets to the actual record length of the record that is read for variable-length sequential files.

Example calls:

```
/* C */                              * COBOL
dbckpt("READ", &io_c, &len, buf);    CALL CP-DBCKPT USING CP-READ,
                                            io-control-area,
                                            io-area-length,
                                            io-area
```

Note: If a file has never been opened, it will be automatically opened on the first READ call. This feature is for compatibility with the mainframe product, and new applications are encouraged to explicitly open files since there is no way to check the return code of the automatic open.

Warning: To retain compatibility with the mainframe API, if a file is already open for writing and a READ call is made, the checkpoint API closes the file and reopens it for reading. This feature was designed to allow certain applications to easily handle temporary files. In general, use of this feature is not recommended in new applications. If this behavior is exploited, the application must take a