

New Technical Notes

Macintosh



Developer Support

'ckid' Resource Format

Platforms & Tools

Written by: Keith Rollin

April 1990

This Technical Note describes the 'ckid' resource format used by MPW's Projector. If you are writing an editor or development system, you may wish to allow or disallow file modification based on the information in the resource.

MPW 3.0 and greater implement a source code control system called Projector. Projector manages sets of source code files known as projects. Users are able to check out source code, make modifications to it, and check it back into the project. Source code can be checked out as “modifiable” or “read-only.” Only one modifiable version of a source file is allowed to be checked out at a time. This feature is very useful if you store your files on a server such as AppleShare; anyone else trying to check out the same file can only check it out “read-only,” ensuring that two people are not modifying the same file at the same time.

Note: This is an overly simplistic description of Projector. It can actually manage more than just source code and has provisions for variations on the modifiable or read-only scheme, such as being able to create experimental offshoots of the main source code called branches.

MPW attaches Projector information to checked out files by adding 'ckid' resources to them. This resource contains information such as to what project the file is attached, when the file was checked out, who checked it out, and whether or not it was checked out read-only. If it **is** checked out read-only, the MPW Shell takes note of this, and does not allow the user to edit the file.

If you are working on a development or source code editing system, then you may wish to respect the 'ckid' resource. At the very least, this means not deleting it. When saving changes to an existing file, some applications write the modified data to a temporary file, delete the old file, and rename the temporary file to have the same name as the original. Unfortunately, this deletes the 'ckid' resource, preventing the user from checking their file back into the Projector database. Applications saving files with this technique should transfer the resource over before deleting the old file.

If you want to include more support for the resource, you could prevent the user from making changes if the file is checked out read-only. This can be done by reading the 'ckid' resource and looking at the appropriate fields. The entire format of the 'ckid'

resource is appended to the end of this Note. You should be interested in only four fields of the resource:

| | |
|-----------------------------|--|
| <code>version</code> | This two-byte field holds the version number of the 'ckid' resource. The current version number is 4. The information presented in this Note is valid for this version number only . Any attempt to apply the information presented here to a 'ckid' resource with a different version would be bad. |
| <code>checkSum</code> | This four-byte field holds a checksum to validate the rest of the resource. It is generated by summing all the subsequent longwords in the resource handle, skipping the checksum field itself and any extra bytes at the end that don't compose a longword. |
| <code>readOnly</code> | This two-byte field indicates whether the attached file is checked out for modifications or not. If the file is not checked out for modifications, this field contains a zero. If the file is modifiable, this field is non-zero and contains special version information for Projector. |
| <code>modifyReadOnly</code> | This one-byte field provides a limited override to the <code>readOnly</code> field. Sometimes it is desirable to be able to modify a file that has been checked out read-only. One may want to do this if they have a file checked out read-only, but later decide to make modifications to it and no longer have access to the Projector database to check out a modifiable version. Under MPW, the user can execute the <code>ModifyReadOnly</code> command. This sets the <code>modifyReadOnly</code> field to non-zero, indicating that the file can be edited, even though it is checked out read-only. |

In your application, you may wish to inhibit modifications to a file if it has a 'ckid' resource and has been checked out read-only. In addition, as a convenience to your customers, you may wish to include a `ModifyReadOnly` feature of your own. To do this, you would need to set the `modifyReadOnly` field to non-zero and recalculate the checksum.

The following routines can help perform these functions. `CKIDIsModifiable` takes a handle to a 'ckid' resource and returns `TRUE` if it indicates that the file is modifiable and `FALSE` otherwise. `HandleCheckSum` takes a handle to a 'ckid' resource and returns a calculated checksum.

MPW Pascal

```
TYPE
  CKIDRec          = PACKED RECORD
    checksum:      LONGINT;
    LOC:           LONGINT;
    version:       INTEGER;
    readOnly:      INTEGER;
    branch:        BYTE;
    modifyReadOnly: Boolean;
    { There's more, but this is all we need }
  END;
  CKIDPtr          = ^CKIDRec;
  CKIDHandle       = ^CKIDPtr;

FUNCTION CKIDIsModifiable(ckid: CKIDHandle): Boolean;

BEGIN
  IF ckid = NIL THEN
    CKIDIsModifiable := TRUE
  ELSE
    WITH ckid^^ DO
      CKIDIsModifiable := (readOnly <> 0) |
                           ((readOnly = 0) & modifyReadOnly);
  END;
END;

FUNCTION HandleChecksum(h: Handle): LONGINT;

VAR
  sum:             LONGINT;
  size:            LONGINT;
  p:               LongintPtr;

BEGIN
  sum := 0;

  size := (GetHandleSize(h) DIV SizeOf(LONGINT)) - 1;
  p := LongintPtr(h^);
  p := LongintPtr(ORD(p) + SizeOf(LONGINT)); { skip over first long
                                             (checksum field) }

  WHILE (size > 0) DO BEGIN
    size := size - 1;
    sum := sum + p^;
    p := LongintPtr(ORD(p) + SizeOf(LONGINT));
  END;

  HandleChecksum := sum;
END;
```

MPW C

```
typedef unsigned long uLong;

typedef struct {
  uLong    checksum;
  long     LOC;
  short    version;
  short    readOnly;
  char     branch;
  Boolean   modifyReadOnly;
  /* There's more, but this is all we need */
}
```

```
} CKIDRec, *CKIDPtr, **CKIDHandle;

pascal Boolean CKIDIsModifiable(CKIDHandle ckid)
{
    if (ckid == nil)
        return(true);
    else
        return( ((**ckid).readOnly != 0) ||
                (((**ckid).readOnly == 0) && (**ckid).modifyReadOnly));
}

pascal uLong HandleChecksum(Handle h)
{
    long        size;
    uLong        sum = 0;
    uLong        *p;

    size = (GetHandleSize(h) / sizeof(long)) - 1;
    p = (uLong *) *h;
    p++;
    /* skip over first long (checksum field) */
    while (size-- > 0) {
        sum += *p++;
    }

    return(sum);
}
```

If you wanted to include a `ModifyReadOnly` function, you could use something like the following Pascal fragment:

```
h := CKIDHandle(Get1Resource('ckid', 128));
IF (h <> NIL) & (h^.version = 4) THEN BEGIN
    h^.modifyReadOnly := TRUE;
    h^.checksum := HandleChecksum(Handle(h));
    ChangedResource(Handle(h));
END;
```

'ckid' Resource format

This MPW Rez resource template is for your application's **information** only. It is valid **only** for version 4 of the resource. Please do not write to this resource or create one of your own. If you feel that you need to change fields in the resource, then limit yourself to the `checksum` and `modifyReadOnly` fields, and **only** if the `version` field is equal to 4. This resource format **will** change in the future.

```
type 'ckid'
{
    unsigned longint;        /* checksum */
    unsigned longint LOC = 1071985200; /* location identifier */
    integer version = 4;     /* ckid version number */
    integer readOnly = 0;    /* Check out state, if = 0 it is modifiable */
    Byte noBranch = 0;       /* if modifiable & Byte != 0 then branch was made
                             on check out */
}
```

```
Byte clean = 0,
    MODIFIED = 1;      /* did user execute "ModifyReadOnly" on this file? */
unsigned longint UNUSED; /* not used */
unsigned longint;      /* date and time of checkout */
unsigned longint;      /* mod date of file */

unsigned longint;      /* PID.a */
unsigned longint;      /* PID.b */

integer;               /* user ID */
integer;               /* file ID */
integer;               /* rev ID */

pstring;               /* Project path */
Byte = 0;
pstring;               /* User name */
Byte = 0;
pstring;               /* Revision number */
Byte = 0;
pstring;               /* File name */
Byte = 0;
pstring;               /* task */
Byte = 0;
wstring;               /* comment */
Byte = 0;
};
```

Notes

The branch field (field 5) holds the letter of this branch (i.e., “a”, “b”, “c”, etc.). It holds zero if this revision is on the main branch.

PID is the Project ID. It is generated using a combination of the tick count and time on your computer in a way that should be sufficient to generate unique Project IDs for every project ever created.

The pstring and wstring fields are variable length fields (pstring is a string preceded by a length BYTE, while wstring is a string preceded by a length WORD), which means that you cannot directly represent this resource with a RECORD in Pascal or struct in C.

Further Reference:

- Macintosh Programmer’s WorkShop 3.0 Reference, Chapter 7,
Projector: Project Management