The DisplayLog Library

Martin Minow
Apple Computer Inc.

INTRODUCTION

The Log Library allows you to add event tracing to all types of Power PC code segments running on PowerMacs that support the Driver Services and Name Registry Libraries, including device drivers, callback routines, dynamically-loaded code segments, and applications. Except for the initialization routine, all functions may be called at any time, even within interrupt or I/O completion routines. The library is designed to be “fail-safe:” it has no error conditions and, barring a malicious attack or dumb bug, will either do what is requested or do nothing, but it should never crash or stall your program. In particular, if your application tries to log data when no LogRecord has been established, or there is no room in the LogRecord data area, your program will not stall, crash or damage the system.

DisplayLog is based on the Audit library available for Macintosh Toolbox application and driver development. DisplayLog, however, runs only on the systems that support the new I/O architecture, in particular the first-generation PCI Power Macintosh computers. The library may be linked directly into Power PC native code segments or loaded from a shared library. When compiled in 68000 emulation, it builds a “glue” library that calls the native shared library. It has not been adapted for Copeland, but this should not be difficult as it only uses operating system services that are forward-compatible with Copeland. (This might not be true for the MacsBug dcmd, of course.)

USAGE

To use the library, you must include its definition file, create a LogRecord, store data, and read data. The following examples show this in the context of a single application; however real-world users would probably split the functions among separate applications; perhaps by creating the LogRecord in an init or driver initialization routine, storing data in a driver or callback code-segment, and displaying data in a stand-alone application or MacsBug dcmd.

Global Variable Definitions

A very simple DisplayLog display application might define the parameters it needs as global values:

```
#include "LogLibrary.h"
LogRecordPtr        gLogRecordPtr;      /* Returned by MakeLogRecord     */
LogEntryRecord      gCurrentEntry;      /* Returned by ReadLogEntry          */
```

Creating a Log Record

Using the above definitions, the following statements create and initialize a DisplayLog record:

```
gLogRecordPtr = MakeLogRecord("SampleLog", 64);
```

The above call creates a log record named "SampleLog" that can store 64 entries. Logging is initially enabled. If there is no room in the record, the first entry waiting for the display routine will be used for this entry (i.e., only the last 64 entries will be retained). Note that MakeLogRecord may only be called from applications or code-segments that can allocate memory in the system resident pool. In a driver, this generally means that it must be called from the initialization routine.

Writing a Log Record Entry

The following code sequence shows how the library might be used to log status errors. It would be called immediately after calling a library function. In the example, the application tries to read data from a file and logs any errors that are returned:

```
status = FSRead(refNum, &bytesToRead, buffer);
if (status != noErr && status != eofErr) {
        gLogRecordPtr = GetLogRecordPtr("Sample");
        LogStatusString(
            gLogRecordPtr ,
            'Read',
            status,
            "\pRead error"
        );
        WriteLogEntry(
            gLogRecordPtr ,
            'Read',
            LogFormat4(
                    kLogFormatSigned,
                    kLogFormatUnsigned,
                    kLogFormatAddress
                    kLogFormatString
                ),
                (signed long) refNum,
                bytesToRead,
                buffer,
                "\pFSRead"
            );
        }
    }
```

This example shows two calls: the LogStatusString macro stores the status code and a string that identifies the function that called it, while the general WriteLogEntry call stores four values: the device reference number, number of bytes, buffer address, and a labelling string.

Reading a Log Record Entry

A typical display application calls ReadLogEntry each time within its event loop, even if no event was returned:

```
for (i = 0; i < 10; i++) {
    if (ReadLogEntry(gLogRecordPtr , &gCurrentEntry) != noErr)
        break;         /* Nothing more to display */
    MyProcessThisLogEntry();
}
```

Here, each pass through the event loop tries to process several entry records, exiting when no more data is available or a reasonable number have been processed. Processing entries in a loop such as the one shown above may prevent a run-away asychronous process from absorbing the entire machine by saturating the log.

Your display application would process each LogEntryRecord entry by formatting the data, including the timestamp, id code, and data. Library routines are available to give a consistent format. Since each entry is self-formatting, a general display routine can be written that will format any records.

Displaying the DisplayLog Entry Data

The following functions convert the DisplayLog entry record into readable Pascal strings.

- LogConvertTimestamp converts the timestamp information from a system-specific up-time value to civil time. Because the internal function, UpTime, is not publicly available to 68000 modules, and the Macintosh Toolbox function, GetDateTime is not available to Power PC functions that use the more limited Driver Services library, this must be called using the code sequence in the example below. LogConvertTimestamp uses a 64-bit integer math package to convert AbsoluteTime to civil time. The function will return an incorrect value if the user changes the base time using the Date Time Control Panel after the function is called.

- FormatLogEntryTimestamp formats the ConvertLogEntryTimestamp values, storing the time in a Pascal string buffer. The time is given in ISO-standard date format as "yyyy.mm.dd hh.mm.ss.msec." This is a fixed-length string whose format is independent of the user's time and date formatting choice. This function does not use the Macintosh Toolbox.

- FormatLogEntryData converts the data portion of the entry record, storing it as a Pascal string. It does not use the Macintosh Toolbox.

For example, the following C sequence may be used to display an entry record:

```
void
MyProcessLogEntry(void)
{
        LogEntryRecord      thisLogEntry;
        DateTimeRec         thisDateTime;
        UInt32              residualNanoseconds;
        Str255              timestamp;
        Str255              content;

        if (ReadLogEntry(gLogRecordPtr , &thisLogEntry)) {
            LogConvertTimestamp(&thisLogEntry, &thisDateTime, &residualNanoseconds);
            timestamp[0] = 0;
            FormatLogEntryTimestamp(timestamp, &thisDateTime, residualNanoseconds);
            FormatLogEntryData(&thisDateTime, content);
            DrawString(timestamp);
            DrawString("\p. ");
            DrawString(content);
        }
}
```

The source of FormatLogEntryTimestamp and FormatLogEntryData should be consulted to see how to process the entry data, including converting the timestamp and "decompiling" the formatted data.

LOGRECORD DATA

The DisplayLog library uses a private data area in the System Resident Pool to store its information. MakeLogRecord and GetLogRecordPtr return a pointer to that area, and all other functions use that pointer as a parameter. The contents of the area are private: your application should not access this record directly,. This is important to prevent asychronous calls (from interrupt routines, for example) from accessing the data simultaneously. The format of the record is, however, in the header file.

Your display application may need to understand the format of the LogEntry record that is returned by ReadLogEntry. This contains the information that was stored as a result of a WriteLogEntry call.

LOGENTRY CONTENTS

ReadLogEntry. if successful, returns a LogEntryRecord. This is a C structure with the following format:

```
typedef struct LogEntryRecord{
      AbsoluteTime            eventTime;  /* UpTime() at call     */
      UInt32                  sequence;   /* Unique entry index   */
      OSType                  idCode;     /* Why are we logging   */
      UInt32                  format;     /* Format of the data   */
      UInt32                  data[10];   /* 40 bytes of data     */
} LogEntryRecord, *LogEntryRecordPtr;
```

The structure elements are used as follows:

`eventTime`	This value timestamps each entry: it is the value of UpTime when the element was written into the log.
`sequence`	This is a unique index for each entry. Gaps in the sequence indicate that data was missed.
`idCode`	This value is the idCode parameter when this record was stored. Your application may use it for any purpose. By convention, it is an OSType (four character string) that further identifies the entry call.
`format`	This defines the format of the rest of the record.
`data`	This contains the actual data. There is enough space for up to ten longwords. Some format parameters store a string which can take up the remaining space. For example, if the only parameter is `kLogFormatString,` up to 39 bytes (plus a one-byte length code) may be stored.

While processing this data is not especially tricky, you may wish to re-read the sample code to unserstand how to format the data values.

LOG RECORD CONTENTS

The DisplayLog library keeps all information it needs in private structures. *It is essential to understand that user applications must not modify these structure directly: they are described here for developers who need to modify the library for their own specific purposes.*

```
typedef struct LogRecord {
      volatile UInt32   semaphore;        /* Atomic access flag   */
      volatile UInt32   lostLockCounter;  /* Atomic lock lost     */
      volatile UInt32   sequenceCounter;  /* Next record sequence */
      volatile UInt32   flags;            /* User configuration   */
      LogRecordNameBuf  logName;          /* System Registry Name */
      volatile UInt32   entryPutIndex;    /* Buffer write index   */
      volatile UInt32   entryGetIndex;    /* Buffer read index    */
      volatile UInt32   entryMaxIndex;    /* Entry max index      */
      LogEntryRecord    entries[1];       /* Entries stored here  */
} LogRecord, *LogRecordPtr;
```

The structure elements are used as follows:

`semaphore`	In order to store data without blocking other programs, all access to the LogRecord is controlled through a "test and set" semaphore that will be locked before reading or writing data to the LogRecord.
`lostLockCounter`	This records the number of times the semaphore was locked (and data consequently lost). Note that this is a different "lost cause" than is recorded by lostDataCounter
`sequenceCounter`	This records the sequence of records added to the log. The first record will have number one. This value will be incremented even if data cannot be stored because the log was full or a semaphore interlock prevents storing an entry. The current value of the sequence counter will be stored into each log entry; gaps in the sequence thus indicate missing data.
`logName`	The name of this LogRecord. This is identical to the property name stored in the System Name Registry. Having a redundant copy simplifies the display application and MacsBug dcmd.
`flags`	This word contains two flag bits: kLogEnableMask and kLogPreserveFirstMask. The flags variable is stored as a longword so that all record elements are aligned to 32-bit

	boundaries: this improves performance on modern hardware.
`entryPutIndex`	This is the ring-buffer index used to store data in the entry vector.
`entryGetIndex`	This is the ring-buffer index used to retrieve data in the entry vector. The case “entryPutIndex equals entryGetIndex” indicates an empty buffer.
`entryMaxIndex`	When the LogRecord is created, this element receives the system ticks value.
`entries[]`	This is a vector of LogEntryRecord elements that the DisplayLog function uses to store data. There is one entry more than the number provided in the call to serve as a “full versus empty” marker.

MakeLogRecord and GetLogRecordPtr return a pointer to the LogRecord.

FUNCTION USAGE

The LogLibrary contains functions to create LogRecords, store data in a LogRecord, and extract data from a LogRecord. In addition, a number of functions that access records within a LogRecord are provided so the library can evolve without requiring application program modification and, especially, so that the library modifies its internal structures in a way that prevents two asychronous requests from changing a structure element at the same time.

Creating a LogRecord

```
      pascal LogRecordPtr
      MakeLogRecord(
            const LogRecordNamePtr  logRecordNamePtr,
            unsigned short          nEntries
      );
```

MakeLogRecordcreates a new LogRecord as a non-relocatable object in the System Resident Pool that is identified by the logNamePtr (a C-string). The record can store the indicated number of LogRecord entries. If successful, logging is initially enabled and preserves the latest entries. MakeLogRecordreturns a pointer to the LogRecord it created or found, or NULL if it cannot create a record. Several independent LogRecords can be active at any time, subject only to memory limitations and good taste.

Because of the design of the Name Registry, LogRecords cannot be named "name."

Finding an Existing LogRecord

```
      pascal LogRecordPtr
      GetLogRecordPtr
            const LogRecordNamePtr  logRecordNamePtr
      );
```

GetLogRecordPtr returns a pointer to the named LogRecord, or NULL if no record is defined for this selector. Your application would typically call this function when it starts since, if a LogRecord is present, it will remain until your system is shutdown or restarted. The value returned by GetLogRecordPtr is used as a parameter to all other functions. Although your application could call GetLogRecordPtr each time it tries to log some data, this requires an internal call to the System Name Registry which would be inefficient.

Writing a LogRecord Entry

```
      OSErr
      WriteLogEntry(
            LogRecordPtr            logRecordPtr,
            OSType                  idCode,
            unsigned long           format,
            ...                     /* Additional parameters, if any */
      );
```

WriteLogEntry stores an entry in the LogRecord that contains the following data:

- WriteLogEntry timestamps each entry with the system UpTime() value. As will be seen, your display procedure, running under the Macintosh Toolbox, can convert this value into civil time (hours, minutes, seconds) when the entry is displayed.

- idCode is a value that you specify. By convention, it is assumed to be an `OSType` that the display application uses to identify the WriteLogEntry request. This may be a unique value for each request or a value that is common to a group of requests (such as a single function or function library). WriteLogEntry does not interpret this value, however FormatLogEntryData interpretes it as an `OSType`. Note that, on the Macintosh, an OSType can be coerced to any longword scalar, such as a memory address. As you adapt the DisplayLog library to your own use, you may wish to consider passing some state information in the idCode parameter (perhaps a task-specific handle) rather than using it as a human-readable identifier.

- format is a 32-bit value that the FormatLogEntryData function interprets to understand how to process the additional parameters, if any. It should be set to the result of expanding the LogFormat macro, or one of two special values. It will be described below.

- Each log entry may store up to ten longword arguments. The format parameter defines their semantic content.

Note: because of differences between various C compilers, it is essential that your program specify all data parameters as longwords. For example, if you wish to log a value that is normally stored as a short value, such as a system error code or Boolean, your program must explicitly cast it to long or unsigned long. If you don't do this, you will display incorrect data and may cause the library to crash. The DisplayLog library operates compatibly under MPW, MetroWerks, and Think C. An application written in one environment can access a LogRecord created by the other environment.

Of course, WriteLogEntry will only store an entry if several conditions are met:

- The logRecordPtr must be non-NULL. It is the value returned by GetLogRecordPtr or MakeLogRecord. If it is NULL, WriteLogEntry silently returns.
- Logging must be enabled.
- There must be room for the entry in the LogRecord entry area. This test will fail if your display function does not read LogRecord entries quickly enough and will be discussed further.

WriteLogEntry returns one of the following status values:

noErr	The entry was stored. This is also returned if the entry could not be stored because the LogRecord was not present or logging is disabled.
writErr	The entry was not stored because the LogRecord was full and "preserve first" selected.
fbsyErr	The entry was not stored because the interlock semaphore indicated that the LogRecord is currently being updated by an asynchronous process.

WriteLogEntry may only be called from C. Pascal callers must use StoreLogEntry.

Formatting LogEntry parameters

In order to simplify display applications, each LogRecordEntry is self-formatting. The format parameter specifies the format of all additional parameters. The format parameter should be constructed by expanding the LogFormat macro. For most uses, you would use LogFormat1, LogFormat2, etc. which create the correct sequence.

The following, if present, must be the last — or only — LogFormat argument specification:

`kLogFormatString`	This must be the last parameter in a LogFormat specification. WriteLogEntry requires one additional parameter, a Pascal string. If it is the only parameter, the first 39 bytes of the string will be stored. If other parameters preceed this, as many bytes as can fit will be stored (each argument requires four bytes). Your program does not need to concern itself with string length; the DisplayLog function truncates the data as needed. Note: The function will properly handle NULL arguments, but may crash if an illegal address is passed.

The following may appear anywhere in the LogFormat operation. Each argument defines the format of an associated WriteLogEntry parameter:

`kLogFormatSigned`	A signed integer longword value.
`kLogFormatUnsigned`	An unsigned integer longword value.
`kLogFormatHex`	An unsigned hexadecimal value that may be interpreted as a 4-byte character, such as an OSType or ResType value. This is displayed both in hexadecimal and as a character string (with '.' replacing any non-USASCII bytes).
`kLogFormatAddress`	An unsigned hexadecimal value that is never interpreted as a character string.
`kLogFormatEnd`	This terminates the list of parameters. The LogFormat1, etc. macros append this as needed: by using these macros, you do not need to be concerned with this value.

In order to simplify the life of the poor programmer, the header file provides a family of macros that allow you to specify one to eight arguments. Thus, instead of writing

```
        WriteLogEntry(
            gLogRecordPtr,
            'Moof',
            LogFormat(
                kLogFormatSigned,   kLogFormatEnd,   kLogFormatEnd,   kLogFormatEnd,
                kLogFormatEnd,   kLogFormatEnd,   kLogFormatEnd,   kLogFormatEnd,
                kLogFormatEnd,   kLogFormatEnd
            ),
            (long) 12345
        );
```

you can write

```
        WriteLogEntry(
            gLogRecordPtr,
            'Moof',
            LogFormat1(kLogFormatSigned),
            (long) 12345
        );
```

Note that the function call explicitly casts the integer parameter to long.

The header file provides `LogFormat1` through `LogFormat10` macros.

Writing a String to the LogRecord

```
        OSErr
        LogString(
                LogRecordPtr            logRecordPtr,
                OSType                  idCode,
                ConstStr255Param        string
        );
```

The LogString function stores a string in a single LogRecord entry. It is actually implemented as a macro as follows:

```
        #define LogString(logRecordPtr, idCode, string) (     \
                WriteLogEntry(                                \
                (logRecordPtr),                          \
                (idCode),                                \
                DLogFormat1(kLogFormatString),                 \
                (string)                                 \
            )                                            \
        )
```

An explicit function form is also available:

```
        (LogString)(gLogRecordPtr, 'ABCD', "\pFoo")
```

Writing a Status Error to the DisplayLog Record

```
        OSErr
        LogStatusString(
              LogRecordPtr              logRecordPtr,
              OSType                    idCode,
              OSErr                     status,
              const StringPtr           string
        );
```

The LogStatusString function stores an operating-system status code and accompaning descriptive string in a single LogRecord entry. LogStatusString is implemented as a macro that calls WriteLogEntry with the proper format expression. An explicit function is also available.

Storing a LogRecord Entry

```
        OSErr
        StoreLogEntry(
              LogRecordPtr              logRecordPtr,
              LogEntryPtr               logEntryPtr
        );
```

StoreLogEntry stores an entry in the LogRecord. This function is normally called from WriteLogEntry, but it may be called directly by Pascal programs (and other languages that do not support C variable-length argument lists). It stores the event time and sequence counter into caller's LogEntry record and, if possible, stores the datum into the LogRecord. This is the only function that stores entries into the LogRecord. It returns one of the following status values:

noErr	The entry was stored. This is also returned if the entry could not be stored because the LogRecord was not present or logging is disabled.
writErr	The entry was not stored because the LogRecord was full and "preserve first" selected.
fbsyErr	The entry was not stored because the interlock semaphore indicated that the LogRecord is currently being updated by an asynchronous process.

Reading a LogRecord Entry

```
        OSErr
        ReadLogEntry(
              LogRecordPtr              logRecordPtr,
              LogEntryPtr               thisLogEntry
        );
```

If there is a LogRecord entry waiting for this LogRecord, ReadLogEntry copies it to your entry buffer and returns noErr. If nothing is waiting (or LogRecordPtr is NULL), it returns readErr. Note that ReadLogEntry does not care whether logging is currently enabled.

If ReadLogEntry returns one of the following status values:

noErr	The entry was retrieved and stored.

readErr	The entry was not retrieved, etther because the LogRecord does not exist or no data is waiting to be stored.
fbsyErr	The entry was not stored because the interlock semaphore indicated that the LogRecord is currently being updated by an asynchronous process.

Formatting the DisplayLog Record Entry Data

```
void
FormatLogEntryData(
      const LogEntryPtr       logEntryPtr,
      StringPtr               result
);
```

FormatLogEntryData converts the data in the entry to a readable string, storing the formatted output in the result string. It does not display the data, nor does it convert the timestamp. The FormatLogEntryData function is provided in LogFormat.c.

Formatting the Log Entry Timestamp

```
void
LogConvertTimestamp(
      const LogEntryPtr       logEntryPtr,
      DateTimeRec             *eventDateTime,
      UInt32                  *residualNanoseconds
);
void
FormatLogEntryTimestamp(
      StringPtr               result,                                    const
DateTimeRec                   *eventDateTime,                            UInt32
      residualNanoseconds
);
```

LogConvertTimestamp converts the entry UpTime value to the civil date and time, storing the result in, eventDateTime, and residualNanoseconds. This function requires the Macintosh Toolbox. The first time it is called, it calls GetDateTime and UpTime to determine the offset between AbsoluteTime and civil time. This means that, if the user changes the system civil time after LogConvertTimestamp is called, subsequent calls will return an incorrect value. LogConvertTimestamp must be linked with the Math64.c library. This function can be called from a 68000 compilation as it uses a cross-architecture call to determine UpTime.

FormatLogEntryTimestamp converts the entry's timestamp to a readable string, storing the formatted output in result. The output is a fixed-length string with the following format:

```
1992.12.25 22:10:44.123
```

The result contains the year, month, and date (in ISO-date format), followed by the time of day in 24-hour clock format with the residual clock ticks (in milliseconds). This format is independent of the time and date formatting selected by the computer user. The FormatLogEntryTimestamp function is provided in LogFormatTimestamp.c.

Support Functions

Your application should use the following functions to access a DisplayLog record's internal structure. They also make it possible to extend the library without requiring changes in an application program. All LogRecord structure elements must be accessed through these functions as they prevent two interrupt-driven code sequences from accessing structure elements simultaneously.

Enabling and Disabling DisplayLog Logging

```
Boolean
EnableLogRecord(
      LogRecordPtr            logRecordPtr,
      Boolean                 enableLogging
);
```

If enableLogging is TRUE, calls to WriteLogEntry will store data in the LogRecord (assuming, of course, that there is room to store data). If the parameter is FALSE, WriteLogEntry ignores calls for this LogRecord. EnableLogRecord returns the old value of the enabling flag; thus your application can restore the previous LogRecord logging state.

Controlling Data Overrun

```
Boolean
PreserveLogRecord(
      LogRecordPtr            logRecordPtr,
      Boolean                 preserveFirst
);
```

If preserveFirst is TRUE and WriteLogEntry is called when there is no room to store data, WriteLogEntry will ignore the call request, preserving the entries waiting to be displayed. If preserveFirst is FALSE, WriteLogEntry will throw away the first (earliest) entry in the “to be displayed” queue, thus retaining the latest nEntries worth of data. PreserveLogRecord returns the old value of the preserveFirst flag; thus your application can restore the previous state. Note that there is no “right” value for this flag: different applications require different preservation strategies.

Getting the Semaphore Lost Counter

```
UInt32
GetLogSemaphoreLostCounter(
      LogRecordPtr            logRecordPtr,
);
```

The semaphroe lost counter is incremented whenever a function cannot access the LogRecord because another asynchronous application is using it. This is unlikely and indicates that data was lost.

Enumeration Utilities

```
OSErr
LogRecordIterateCreate(
      LogRecordIterPtr        cookie
);
void
LogRecordIterateDispose(
      LogRecordIterPtr        cookie
);
LogRecordPtr
LogRecordIterate(
      LogRecordIterPtr        cookie
);
```

These routines allow an application to retrieve all known LogRecords. The iteration process is as follows:

```
if (LogRecordIterateCreate(&cookie) == noErr) {
   while ((logRecordPtr = LogRecordIterate(&cookie)) != NULL) {
   ... Process this log record, the name is in the record ...
   }
}
LogRecordIterateDispose(&cookie);
```

These functions can be called from 68000 compilations by using LogLibrary68.c.

PERFORMANCE TIMING

Because the LogLibrary timestamps all entries, it is fairly easy to use it to instrument device drivers. To test a driver, you can define a Timestamp macro as follows:

```
#define Timestamp(tag) do {                                     \
      WriteLogEntry(gLogRecordPtr, tag, LogNoFormat);           \
   } while (0)
```

Then, you could bracket interesting sequences as follows:

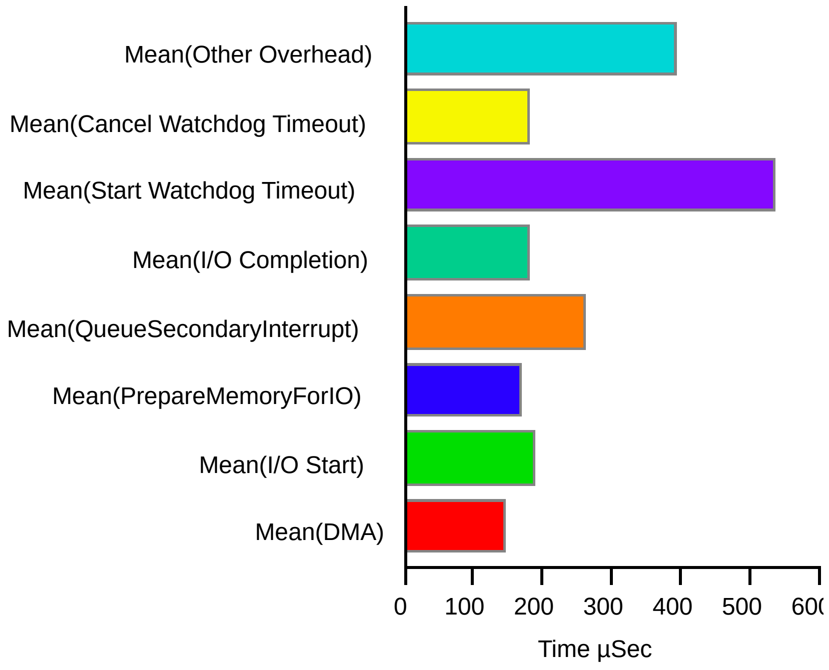
```
Timestamp('XFE+');            /* Overall transfer start          */
PBWriteSync(&TEST.pb);
Timestamp('XFE-');            /* Overall transfer ends           */
```

After each transfer, the log was read and sequence times computed using the following algorithm:

```
const OSType gTagNames[sMaxStatistics] = {
  'XFE ','DMA ','IOS ','PMI ','Q2I ','IOC ','SWT ','CWT ','SSI '
};

while (ReadLogEntry(gLogRecordPtr, &logEntryRecord) == noErr) {
  switch (logEntryRecord.idCode & 0x000000FF) {
  case '+':  isStart = TRUE; break;
  case '-':  isStart = FALSE;  break;
  default:   continue;
  }
  eventTag = (logEntryRecord.idCode & 0xFFFFFF00) | ' ';
  for (i = 0; i < sMaxStatistics; i++) {
      if (eventTag == gTagNames[i]) {
      if (isStart)
          eventStart[i] = logEntryRecord.eventTime;
        else {
          eventNanoseconds = AbsoluteDeltaToNanoseconds(
                 logEntryRecord.eventTime,
                 eventStart[i]
             );
        eventVector[i] += eventNanoseconds.lo;
          eventStart[i].lo = eventStart[i].hi = 0;
      }
      }
    }
}
```

The test program then wrote the results to a file where they were processed by a statistics package to yield the following chart:


Mean(Other Overhead)
Mean(Cancel Watchdog Timeout)
Mean(Start Watchdog Timeout)
Mean(I/O Completion)
Mean(QueueSecondaryInterrupt)
Mean(PrepareMemoryForIO)
Mean(I/O Start)
Mean(DMA)
0
100
200
300
400
500
600
Time µSec