4D Advanced Debugging Techniques – Part 2

By Hugo Fournier, Technical Support Manager, 4D, Inc.

TN 06-33

Abstract

本テクニカルノートは次のような二部構成になっています：

- 4th Dimension (4D) 2004 に組み込まれているデバッグツールに関する記事。(第一部)
- テキスト形式のログファイルを出力し、4D 2004 に組み込まれているデバッグツールでは対応が困難な分野をデバッグする方法に関する記事。(第二部)

第一部では、デバッグ作業の基本コンセプト、および 4D に組み込まれているデバッグツールについて解説した後、少し高度なデバッグテクニックをいくつか紹介しました。

第二部では、ログファイルの出力を中心としたデバッグの基本コンセプトについて解説した後、汎用的なログ出力コンポーネント、そのインストール方法および使用方法について説明します。

Introduction

第一部では、デバッグの基本コンセプトおよび 4D との関係が取り上げられました。
その中で、デバッグの過程は次のようなステップに整理できると述べました：

1. バグの存在を認める。
2. バグ発生の再現パターンを特定する。
3. バグの出所を特定する。
4. バグの原因を特定する。
5. バグの修正(フィックス)を開発する。
6. バグフィックスを適用する。
7. バグフィックスをテストする。

4D には、次のようなデバッグツールが用意されているとも述べました：

- デバッグ/トレースウインドウ
- ランタイムエクスプローラ
- デバッグログファイル
- エラーメッセージ

What if 4D cannot help?

上記のツールは、デバッグログファイルを除き、いずれもバグの発生パターンを突き止める上ではほとんど無力であることを認めなくてはなりません。それらのツールは、基本的に再現パターンが分かっているバグの出所を特定することに目的が絞られています。

第一部では、順を追ってデバッグをすることが勧められていました。それによれば、再現のパターンの特定は、バグの出所を特定するよりも前に済んでいなければならないことになります。記事では、再現パターンの発見が全行程の中でもっとも重要かつ時間のかかる作業であるとも述べられていました。

最後に、デバッグツールの大部分はコンパイルされたデータベースでは利用できないため、コンパイルモードのみで発生するバグについては、打てる手が非常に限定されているという点も指摘されていました。

こうした問題は、カスタムログファイルを生成することによって回避できるかもしれません。

Advanced Debugging Part 2 – Logging

4D に用意されているデバッグオプションでは目的が達成できない場合の代替案を紹介します。

Introduction

カスタムログファイルを出力することの狙いは、他のデバッグツールにおける制限事項を回避することにあります。たとえば、ログファイルは前述したデバッグ手順のステップ 2、バグの再現パターンを突き止める段階で役立ちます。バグが特定のイベント実行順序やタイミングに起因しているような場合、ログファイルがあれば、パターンが特定できるかもしれません。

ログファイルの別の利点は、コンパイルモードでも問題なく使用できるという点です。

まとめると、デバッグツールとしてのカスタムログファイルには次のような特徴があります：

- パターンの解析。
  - 解析できる。（例：テキストエディタ、スプレードシートなどを使用。）
  - ログ同士を比較できる。
- タイミング/実行順序に起因する問題を検出。
- インタプリタモード/コンパイルモードで動作。

こうしたアドバンテージについては、続くセクションで詳細にわたって取り上げられます。

Pattern Recognition

ログファイルは、パターンを検出する際にその効果を発揮します。以下は簡単な例です：

```
Handle Creation.     Errhp: 135853116
Handle Creation.     Errhp: 135890816
Handle Creation.     Errhp: 135888944 <- リーク
Handle Free.                 Errhp: 135890816
Handle Free.                 Errhp: 135853116
Handle Creation.     Errhp: 136015956
Handle Creation.     Errhp: 136014084 <- リーク
Handle Free.                 Errhp: 136015956
```

上記ログファイルは、4D for OCI プラグインを使用するデータベースで生成されたものです。ファイルは 4D では倍長整数で返される 4D for OCI の「エラーハンドル」の作成およびクリアの記録が含まれています。

この簡単な例から明らかなように、メモリの割り当てを記録するだけで簡単にメモリーリークを発見することができます。上記の例では、解放されないエラーハンドルがログにしっかり残っています。

テキスト形式(標準テキスト、XML、HTML を含む)ログファイルのもうひとつのメリットは、テキストエディタやスプレッドシートのような他のアプリケーションで開いて解析することができるという点です。デベロッパはファイルが難解になることを気にすることなく、望むだけ多くの情報を記録することができます。後で必要な情報だけをフィルタすれば良いからです。ファイルの解析は大変な作業になるかもしれませんが、他の方法では決して知り得ないパターンが発見できる場合が少なくありません。

最後に、データベースの実行記録が残るという点も見逃せません。たとえばデバッガの使用と比較した場合、一方は処理を続行した場合、あるいはデータベースを閉じた時点で実行記録が失われてしまいます。ログファイルはアーカイブすることができ、後で比較検証することができます。繰り返しになりますが、他の方法では決して知り得ないパターンが発見できるかもしれません。

Detecting Problems Related to Timing/Synchronization

ログファイルが威力を発揮する別の分野は、実行のタイミングに起因する問題の分析です。デバッガは(ブレークポイントやTRACEコマンドでプロセスを停止することにより)実行タイミングが大幅に変わってしまうため、実際の動作タイミングを再現することにはなりません。ログファイルを出力するのであれば、アプリケーションは通常どおり*に実行され、動作タイミングは後でログファイルを調べることにより解析することができます。

以下に簡単な(そして若干恣意的な)例を紹介します。このサンプルでは、インタープロセス変数 <>n の階乗を計算するメソッドが再帰的に実行されており、途中、別のプロセスが<>n の値を更新するというシチュエーションを仮定しています。

```
PID<6> (0)Start Factorial of: <>n = 20
PID<7> <>n の値を 4 に変更してしまったために問題発生...
PID<6> (0)Before:      test_Factorial( 4 )
PID<6> (1)   Method Start: test_Factorial
PID<6> (1)   Before:      test_Factorial( 3 )
PID<6> (2)      Method Start: test_Factorial
PID<6> (2)      Before:      test_Factorial( 2 )
PID<6> (3)        Method Start: test_Factorial
PID<6> (3)        Before:      test_Factorial( 1 )
PID<6> (4)          Method Start: test_Factorial
PID<6> (4)          test_Factorial done!
PID<6> (4)          Method End:    test_Factorial
PID<6> (3)        After:   2:=test_Factorial( 1 )
PID<6> (3)        Method End:    test_Factorial
PID<6> (2)      After:   6:=test_Factorial( 2 )
PID<6> (2)      Method End:    test_Factorial
PID<6> (1)   After:   24:=test_Factorial( 3 )
PID<6> (1)   Method End:    test_Factorial
PID<6> (0)After:   24:=test_Factorial( 4 )
```

結果 4! = 24

20!=24 ではありませんが、途中で<>n が更新されたために結果が変わってしまいました。ログファイルにはプロセス ID が記録されているので、何が起きたのかを理解することができます。

* ログファイルの出力には余分の処理が必要なので、厳密にいえば通常の動作ではありません。ログ出力を追加した結果、問題の質が変化したり、問題が起きなくなってしまうようであれば、この点を考慮する必要があります。それでも、ログの出力によって問題が変化するということ自体、それがタイミングに起因する現象であることを確証するものであり、解決の糸口になるのでやはり有益です。

Interpreted vs. Compiled

ログファイルの出力という方法は、コンパイルされたデータベースにも応用ができるという点でも優れており、インタプリタモードでは問題がなく、コンパイルモードでのみ問題が起きるというような場合にとりわけ有用です。データベースをコンパイルしてしまうと、4D デバッグツールの多くは使用できなくなってしまい、窮地に立たされてしまいます。デバッガ/トレースウインドウは使用できなくなり、ランタイムエクスプローラは機能が制限され、エラーメッセージの大部分は表示されなくなります。もちろん、4D のビルトインデバッグログファイルは引き続き利用することができますが、内容をカスタマイズする方法がなく、特定の情報が含まれているとは限りません。

インタプリタ/コンパイルどちらのモードであっても、カスタマイズされたログファイルは前述の目的、つまり問題発生パターンの検出、タイミングおよび同期に関連した問題の特定のために有用なツールです。

Conclusion

適切な状況で使用すれば、ログファイルによるデバッグは非常に強力なツールとなります。問題発生のパターンを解析する場合、実行タイミングに起因する問題を調べる上で、とりわけこのテクニックは重要です。ログファイルによるデバッグは、コンパイルモードで利用できる数少ないデバッグツールのひとつでもあります。第一部で取り上げた標準のデバッグツールと合わせて使用すれば、デベロッパはかなりのデバッグ環境が揃えられることになるでしょう。

4D Logger 2004-1 Component Reference

このテクニカルノートに収録されている 4D Logger 2004-1 コンポーネント(以下、Logger)は、あらゆるデータベースでテキスト形式のログファイルを出力するためのものです。コードはオープンソースとして提供されており、利用者は自由にコードを修正することができます。

Important General Notes

このセクションでは、Loggers を開発した際に考慮した基本コンセプトについて説明します。

Stack Level

デバッグメッセージのログでは、しばしば「スタックフレーム」という概念が採用されます。コンポーネントコードでは、同じ概念に「スタックレベル」という表現で言及しています。スタックレベルは、ログファイルの中で加算される整数とスペースによるタブで表現されており、メソッドの開始がログされるたびに、該当するプロセスのスタックレベルが上がり、続くログエントリーはそのレベルで記録されるようになっています。メソッドが終了すると、スタックレベルが下がります。

スタックレベルは、4D のいわゆるスタックとは何の関係もありません。スタックレベルは単純にログを見やすくするためのコンセプトです。

```
PID<1> (0)Method Start: OD Execute SQL
PID<1> (1)        Method Start: OCI_Exist_Login
PID<1> (1)        Method End: OCI_Exist_Login
PID<1> (1)        Method Start: OD Create cursor
PID<1> (2)                Method Start: OCI_Exist_Login
PID<1> (2)                Method End: OCI_Exist_Login
PID<1> (2)                Before: OCIHandleAlloc( 129735744; 0; ...
PID<1> (2)                After: 0:=OCIHandleAlloc( 129735744; 129796272; ...
PID<1> (1)        Method End: OD Create cursor
PID<1> (0)Method End: OD Execute SQL
```

スタックレベルは､特定の条件で自動的に増減します(次の段落をご覧ください)｡デベロッパは、手動でスタックレベルを増減することもできます。

What Gets Logged?

Loggers には 6 種類のログタイプが用意されており、それぞれに適切なフォーマットが存在します。ログ出力をコマンド制御にする目的は、汎用性を高め、実行内容を理解しやすくすることにあります。とはいえ、次のようなコマンドには少し問題あります：

Log_a_message(“Before: OCIHandleAlloc(129735744; 0)”)

内容は明快ですが、値がハードコーディングされているので汎用的な用途に向いていません。

Loggers コンポーネントでは、関数コール、プロシージャコールなどそれぞれにコマンドが存在し、自動的にパラメータが解析されてメッセージが作成されます。結果、4D コードは分かりやすくなり、出力されるログファイルにも統一感があります。

Functions:

関数の場合、Loggers はパラメータ、関数名、返り値を含む情報をコールの前後に記録します。(注記：この記事では返り値のあるコマンドおよびメソッドを関数と呼んでいます。)
スタックレベルは関数コードの前("before")に加算、後("after")で減算され、間に発生したイベントのログエントリーはインデントされます。たとえば、次のように記録されます：

```
PID<7> (3)      Before:     test_Factorial( 2 )
PID<7> (4)          Before:     test_Factorial( 1 )
PID<7> (5)                      test_Factorial done!
PID<7> (4)          After:   2:=test_Factorial( 1 )
PID<7> (3)      After:   6:=test_Factorial( 2 )
```

Procedures:

プロシージャのログは、基本的に関数と同じですが、返り値がないために記録されない点が異なっています。(注記：この記事では返り値のないコマンドをプロシージャと呼んでいます。)

Methods:

メソッドのログは、開始時と終了時に特定のコマンドを実行することで記録されます。ログのフォーマットは、次のようになっています：

PID<プロセス番号> (スタックレベル) インデント Method Start: メソッド名

例：

```
PID<1> (2)      Method Start: OCI_Exist_Cursor
PID<1> (2)      Method End: OCI_Exist_Cursor
```

コマンドは、メソッドの最初と最後に実行されるべきであり、メソッドコールの前後に実行するべきではありません。コマンドは単純にログに記録を追加するためのものであるので、スタックレベルの増減は実施されません。

Messages with stack level:

スタックレベルの増減は自動的に制御しつつ、汎用的なログを残すためのコマンドもあります。

```
PID<1> (3)      Message: Foo
PID<1> (4)      Message: BarP
```

PID<1> (5)　　Message: Baz

Messages without a stack level:

さらに汎用的なコマンドでは、プロセス ID だけが自動的に制御され、スタックレベルも手動で増減するようになっています。

PID<5> プロセス番号 5 から出力された適当なメッセージ

Unformatted messages:

改行以外はすべて手動で完全にカスタマイズできるログメッセージも記録することができます。

What Parameter Types Are Supported?

ログ出力コンポーエントが現在サポートしているのは、整数(INTEGER, LONGINT)および文字(STRING, TEXT)タイプのパラメータです。その他のタイプは<n/a>としてログファイルに記録されます。それでも、ログファイル出力コンポーネントは容易に拡張することができます。コンポーネントメソッド DL_Tool_ParamToString を参照してください。

Debug Log Monitor Process

ディスク容量の問題は、テキストファイルによるログ出力システムの宿命です。メッセージが蓄積してゆけば、ファイルサイズはどんどん大きくなってゆきます。

Logger には、ディスク容量の逼迫を防ぐためにデバッグログファイルを監視する 4D プロセスがあります。注記：このプロセスを使用しない場合、ログファイルはいつまでも大きくなってゆきます。デフォルトの設定で(DL_INIT を実行すれば)、監視プロセスは ON です。

Debug Log Files

デバッグログファイルは、標準テキストファイル形式で出力されます。

ハードディスクの容量を逼迫しないため、デフォルトでログファイルのサイズは 500KB が上限となっています。上限に達したログファイルは、別にアーカイブされる仕組みです。ログファイルの最大サイズは、デベロッパが設定できるようになっています。

デフォルトで最新 6 ファイルまでが残されるようになっています。これもまたハードディスク容量を過度に占有しないための処置です。ファイル数の制限は、特定のセッションごとに適用されます。前回の起動で生成されたログファイルが削除されることはありません。保存するログファイルの最大数は、デベロッパが設定できるようになっています。

いずれの仕組みも、ログ監視プロセスが実行されている場合にのみ動作する点に留意してください。管理プロセスが OFF であれば、際限なくログファイルが大きくなってゆきます。

複数のプロセスによるログファイルの更新を可能にするため、セマフォが用いられています。

ログファイルは、最初のエントリーがあるまで作成されません。DL_INIT を実行した時点では、ファイル名が確定するだけであり、実際にファイルが作成されるのは、最初のログエントリーがあった後になります。ファイルが作成される場所をデベロッパが変更できるようにするため、このような仕組みにしました。

Debug Log Flags

Logger は、フラグによる合図システムを採用しており、必要に応じてログ出力の ON/OFF が切り替えられるようになっています。このシステムの詳細は後述しますが、ポイントは、ログ出力が一括制御できるようになっているという点です。フラグは複数使用することができ、特定のログ項目をグループ化し、独自に ON/OFF を切り替えることができます。たとえば、関数コールのログをグループ化し、診断メッセージとは別のグループにすることができます。

これは任意のオプションであり、他のログコマンドの使用を制限するものではありません。

Internal Members and Methods

Logger 内部の変数は、いずれも直接アクセスされることは想定してはいません。変更してもよい変数については、アクセスするためのメソッドが用意されています。コンポーネントが Public 属性である以上、直接アクセスを妨げるものはありませんが、変数に直接アクセスするのであれば、慎重に操作を実行してください。

メソッド名の接頭辞が DL_Tool となっているメソッドは、基本的に直接コールされることを想定していません。直接コールするのであれば、内容を理解した上、慎重に実行してください。

Important Method Notes

このセクションでは、Logger コンポーエントの主要なメソッドについて説明します。

メソッド : DL_INIT

ログコードの初期化を実行し、監視プロセスを起動するメソッドです。データベースのスタートアップ部分に記述し、ログ出力コードを開始する前に実行するようにしてください。また、コンポーネントがセマフォを確認する間隔(デフォルトで 10 ticks)もこのメソッドの中で定義しています。

メソッド : DL_LOGFILE_NEW

デバッグログファイルのファイル名を変更するには、このメソッドを修正してください。Logger コンポーネントにはファイル名を設定するためのインタフェースはなく、直接インタープロセス変数を変更する以外には、ファイルを変更する方法がありません。

デフォルトのデバッグログファイル名は、次のような形式になっています：

Log Created MM-DD-YYYY hh_mm_ss.txt

メソッド : DL_Tool_ParamToString

ログ出力の際にパラメータが整形される様式を定義するメソッドです。たとえば、20 バイト長を超えるテキスト型のパラメータが DL_Log_FunctionBefore に渡された場合、ファイルをみやすくするためにデータの末尾を切り捨てて...が追加されるようになっています。

整数や文字以外のデータタイプもサポートするようにコンポーネントを改造するのであれば、このメソッドを修正すると良いでしょう。入力値はパラメータに対するポインタ、出力値はファイルに書き出されるテキストです。

メソッド : DL_Constants_@

DL_Constants_ERROR、DL_Constants_OFF および DL_Constants_ON メソッドは、定数のように使用してください。このような仕組みにしたのは、4DK#リソースでカスタム定数を作成するよりも簡単に定数のようなものが使用でき、インタープロセス変数(<>DL_K_ERROR, <>DL_K_OFF, <>DL_K_ON)に対する直接のアクセスを防ぐことができるためです。詳細は後述しましが、いずれのメソッドもフラグによる信号システムで専ら使用します。

4D Logger 2004-1 Command Reference

このセクションは、Logger コンポーネントのコマンドリファレンスドキュメントです。

Themes

Logger のメソッドは、以下のテーマに分類されています。(接頭辞がテーマに対応しています。)

DL_Constants

定数のように扱われるメソッドです。変数に対する直接アクセスは推奨されていません。

DL_Flag

フラグシステム専用のメソッドです。

DL_Get / DL_Set

コンポーネントで使用する変数の値を参照または更新するためのメソッドです。変数に対する直接アクセスは推奨されていません。

DL_LOGFILE

ログファイルに関係したメソッドです。

DL_Log

ログを書き出すためのメソッドです。

DL_Monitor

監視プロセスの開始、停止、設定に関係したメソッドです。

DL_Tool

基本的にデベロッパが直接コールするべきではないメソッド、つまりコンポーネントが内部的に使用しているメソッドです。

Methods

ドキュメントには、デベロッパの観点から価値にあるメソッドの情報だけが含められています。その他のメソッドについては、ソースコードのコメント文を参照してください。

なお、すべてのコンポーネントメソッドには、実行目的および入出力パラメータの記載されたヘッダコメント文が付いています。

DL_Constants_ERROR

<>DL_K_ERROR の値を返します。

DL_Constants_OFF

<>DL_K_OFF の値を返します。

DL_Constants_ON

<>DL_K_ON の値を返します。

DL_Flag_Create

新しいデバッグログフラグを作成します。

DL_Flag_Delete

既存のデバッグログフラグを削除します。

DL_Flag_State

デバッグログフラグの値を参照するために使用します。返される値は、DL_Constants_ON または DL_CONSTANTS_OFF です。フラグが存在しない場合、あるいは他の理由で問題がある場合、DL_Constants_ERROR が返されます。

DL_Flag_TurnOff

ログフラグを OFF に設定します。

DL_Flag_TurnOn

ログフラグを ON に設定します。

DL_Get_LogFileName

カレントログファイルのファイル名を返します。内部的にセマフォを使用しています。

DL_Get_LogFilePath

カレントログファイルのファイルパス名を返します。内部的にセマフォを使用しています。

DL_Get_LogLimit

アーカイブする最大ファイル数を返します。

DL_Get_LogSizeLimit

ログファイルの最大ファイルサイズを返します。

DL_INIT

コンポーネントが使用するインタープロセス変数を初期化するメソッドです。最初に一度、理想はスタートアップメソッドの中で実行するようにしてください。その他の DL_@メソッドは、このメソッドを実行した後に使用するべきです。

DL_LOGFILE_DELETECURRENT

カレントログファイルがあれば削除します。注記：ログファイル名は初期化されません。後続のログ出力コマンドは、同名の新規ログファイルに書き出されます。ログファイル名を変更するには DL_LOGFILE_NEW を実行してください。

DL_LOGFILE_NEW

ログファイル名を定義します。。後続のログ出力コマンドは、新しいファイル名を使用します。コマンドを実行した時点では名前が設定されるだけであり、ファイルは作成されません。

DL_Log_FunctionAfter

コールした関数の実行が完了したことを意味するログメッセージを出力するために使用します。

(注記：この記事では返り値のあるコマンドおよびメソッドを関数と呼んでいます。)

メッセージの形式：

PID<プロセス ID> (スタックレベル)　After:　ReturnVal:=関数名(p1;p2;...;pn)

P1...は関数に渡されたパラメータ。

DL_Log_FunctionBefore

関数を実行しようとしたことを意味するログメッセージを出力するために使用します。

メッセージの形式：

PID<プロセス ID> (スタックレベル)　Before:　関数名(p1;p2;...;pn)

P1...は関数に渡されたパラメータ。

DL_Log_Message

改行付きでログメッセージを出力するために使用します。

DL_Log_MessageNoStack

プロセス ID 付きでログメッセージを出力するために使用します。

メッセージの形式：

PID<プロセス ID>　メッセージ

DL_Log_MessageWithStack

プロセス ID、スタックレベル、インデント付きでログメッセージを出力するために使用します。

メッセージの形式：

PID<プロセス ID> (スタックレベル) インデント　メッセージ

DL_Log_MethodEnd

メソッドの完了を意味するログメッセージを出力するために使用します。

メッセージの形式：

PID<プロセス ID> (スタックレベル) インデント　Method End:　メソッド名

DL_Log_MethodStart

メソッドの開始を意味するログメッセージを出力するために使用します。

メッセージの形式：

PID<プロセス ID> (スタックレベル) インデント　Method Start:　メソッド名

DL_Log_ProcedureAfter

プロシージャが完了したことを意味するログメッセージを出力するために使用します。

(注記：この記事では返り値のないコマンドをプロシージャと呼んでいます。)

メッセージの形式：

PID<プロセス ID> (スタックレベル)　After:　プロシージャ名(p1;p2;...;pn)

P1...はプロシージャに渡されたパラメータ。

DL_Log_ProcedureBefore

プロシージャを実行しようとしたことを意味するログメッセージを出力するために使用します。

メッセージの形式：

PID<プロセス ID> (スタックレベル)　Before:　プロシージャ名(p1;p2;...;pn)

P1...はプロシージャに渡されたパラメータ。

DL_Monitor_Start

ログファイルを監視し、必要に応じてアーカイブするプロセスを起動します。

DL_Monitor_Stop

監視プロセスを停止するために使用します。内部的にインタープロセス変数の値を変更します。

DL_Set_LogFilePath

ログファイルの保存先パスを設定するために使用します。デフォルトパスは、一時フォルダ(Temporary folder 関数の返り値)です。

DL_Set_LogLimit

ログファイルの最大ファイル数を設定するために使用します。デフォルトの数は 6 です。

DL_Set_LogSizeLimit

ログファイルの最大サイズを設定するために使用します。最大サイズに達したログファイルはアーカイブされます。デフォルトのサイズは 500KB です。

その他の Logger メソッドは、コンポーネントの内部で使用されるものです。基本的な情報は、各メソッドのソースコードに記述されています：

DL_Tool_CheckFlagArrays

DL_Tool_CreateLogFile

DL_Tool_Error

DL_Tool_FormatLogFilePath

DL_TOOL_LOGFILE_ARCHIVE

DL_Tool_LogMessage

DL_Tool_MonitorLog

DL_Tool_ParamCountAlert

DL_Tool_ParamToString

DL_Tool_StackDecrease

DL_Tool_StackIncrease

4D Logger 2004-1 Installation Procedures

4D Logger 2004-1 の使用には、4D 2004 が必要です。

Installing the　4D Logger component

1. データベースを 4D Insider で開きます。
2. コンポーネントメニューのインストール／更新を選択します。

3. 4D Logger 2004-1.4CP コンポーネントの場所を指定して開くボタンをクリックします。

4. コンポーネントのインストールが完了したら 4D Insider を終了します。
5. データベースを 4D で開きます。
6. メソッド DL_INIT を On Startup データベースメソッド、あるいはスタートアップ処理の一部に追加します。
7. 4D を終了します。
8. スタートアップが実行されるように 4D を再起動してデータベースを開きます。

(前のステップの最後に DL_INIT を実行しても構いません。)

Using the 4D Logger Component

Basic Usage

インストール作業を終えたならば、特に設定をすることなく、すぐにデフォルトの設定でデバッグログコンポーネントを使い始めることができ、DL_Log テーマのメソッドを実行すれば、ログが出力されます。たとえば、次のようなコードで「Hello World!」というログメッセージが出力されます：

```
DL_Log_Message("Hello World!")
```

ログファイルの設定を変更するには、DL_Set テーマのメソッドを実行してください：

- ログファイルの保存場所
- 保存するログファイルの最大数
- ログファイルをアーカイブする最大ファイルサイズ

ログ監視プロセスの起動および停止には、DL_Monitor テーマのメソッドを実行してください。監視プロセスを停止すると、ファイルサイズが際限なく大きくなるので注意が必要です。

In-Depth DL_Log Examples

DL Log テーマのコマンドには、ポインタで任意のパラメータを渡します。たとえば、文字型のパラメータを受け取る関数があり、その使用をログに記録する場合、次の記述ではいけません。

```
DL_Log_FunctionBefore("someFunction";"ログに記録する文字列")
someFunction ("文字型のパラメータ")
```

そうではなく、ローカル変数を宣言して値をポインタで渡してください：

```
C_TEXT($debug_StringLiteral)
$debug_StringLiteral:="ログに記録する文字列"
DL_Log_FunctionBefore("someFunction";->$debug_StringLiteral)
someFunction ("文字型のパラメータ")
```

ログに記録する文字列をパラメータとしてメソッドに渡すような方法は、あまり推奨されていません。後でデバッグコードを除去するときに修正が大変だからです。たとえば、次のコードはあまり良い例ではありません：

```
C_TEXT($debug_StringLiteral)
$debug_StringLiteral:="ログに記録する文字列"
DL_Log_FunctionBefore("someFunction";->$debug_StringLiteral)
someFunction ($debug_StringLiteral)
```

オリジナルのコードには手をつけないで、デバッグログコードを追加するようなスタイルにしたほうが、後でコードを除去するときに楽です。(実際には、ログに記録する文字列も変数で定義し、データベースにハードコーディングしないほうが良いでしょう。)

除去するときのことを考え、デバッグログコード全体をコメントで強調すると良いでしょう：

```
`Begin Debug Code Block-----------------------------------
DL_Log_FunctionStart(Current method name)
`End Debug Code Block-------------------------------------
```

デバッグコードブロックを囲むコメント文に何らかの法則を設けるなら、後で(たとえば「データベース中を検索」を使用して)コードを除去したり変更したりするのが容易になります。

Function example:

ユーザカスタムプロセスで以下のコードを実行：

```
C_LONGINT($in;$result)
$in:=1
DL_Log_FunctionBefore("test_Factorial";->$in)
$result:= test_Factorial($in)
DL_Log_FunctionAfter(->$result;"test_Factorial";->$in)
```

ログファイルに記録されるメッセージ：

```
PID<1> (0)Before:            test_Factorial( 1 )
PID<1> (0)After:         1:=test_Factorial( 1 )
```

Procedure example:

ユーザカスタムプロセスで以下のコードを実行：

```
C_LONGINT($foo)
$foo:=100
DL_Log_ProcedureBefore("Some_Procedure";->$foo)
Some_Procedure($foo)
DL_Log_ProcedureAfter(Some_Procedure";->$foo)
```

ログファイルに記録されるメッセージ：

```
PID<1> (0)Before:        Some_Procedure ( 100)
PID<1> (0)After:         Some_Procedure ( 100 )
```

Method example:

ユーザカスタムプロセスで実行された SomeMethod メソッドで以下のコードを実行：

```
DL_Log_MethodStart(Current method name)
`メソッドのコード本体
DL_Log_MethodEnd(Current method name)
```

ログファイルに記録されるメッセージ：

```
PID<1> (0)Method Start:    SomeMethod
PID<1> (0)Method End:      SomeMethod
```

Messages with stack level example:

ユーザカスタムプロセスで実行された再帰的メソッドの 5 回目の実行時に以下のコードを実行:

```
DL_Log_MessageWithStack(Current method name+" done!")
```

ログファイルに記録されるメッセージ：

```
PID<1> (5)               test_Factorial done!
```

Messages without stack level example:

ユーザカスタムプロセスで以下のコードを実行：

DL_Log_MessageNoStack("Begin test!")

ログファイルに記録されるメッセージ：

PID<1> Begin test!

Messages with no extra formatting example:

適当な場所で以下のコードを実行：

DL_Log_Message("Hello World!")

ログファイルに記録されるメッセージ：

Hello World!

Using Log Flags (DL_Flag examples)

新しいログフラグを作成します。(デフォルト値は ON です)：

DL_Flag_Create("MyFlag!")

フラグをコードの中で使用してログ出力を制御する方法：

```
If (DL_Flag_State("MyFlag!")=DL_Constants_ON)
    `ログ出力コードを実行
End if
```

ログの状態を調べるために DL_Constants 系メソッドは定数のように使用されている点に留意してください。(フラグの状態は ON、OFF または ERROR です。)

フラグを OFF にする方法：

DL_Flag_TurnOFF("MyFlag!")

フラグを削除する方法：

```
DL_Flag_Delete("MyFlag!")
```

複数のフラグを使用することには、どんな益があるでしょうか。次の例題では、フラグを 2 個、作成し、一方は情報系のログ、他方は実際のデバッグログを制御するために使用しています：

```
C_LONGINT(LogCodeFlag;LogMessFlag)
C_TEXT($p1)
LogCodeFlag:="Log Code"
LogMessFlag:="Log Messages"
$p1:="MESSAGE コマンドに渡すパラメータ..."

If (DL_Flag_State(LogMessFlag)=DL_Constants_ON)
    DL_Log_MessageWithStack("情報: 変数宣言の終わり。")
End if

If (DL_Flag_State(LogCodeFlag)=DL_Constants_ON)
    DL_Log_ProcedureBefore("MESSAGE";->$p1)
End if

MESSAGE($p1)
```

Log Messages フラグが OFF であれば、情報系のログは記録されません。Log Code フラグが OFF であれば、実行コードのログは記録されません。デベロッパはフラグを操作することにより、記録されるログの種類を管理することができます。

Putting it all together

テクニカルノートに収録されているサンプルデータベース(Logger_Test.4DB)は、すべてのログ出力メソッドを実行するコードが含まれています。小さなデータベース(おそらくコンポーネントのほうが大きい)ですが、すべての機能がテストできるように設計されています。

A Final Word

4D Logger コンポーネントではカバーできていない部分もありますが、完全なソースコードが提供されているので、用途に合わせて自由に修正してください。デベロッパがスムーズにコードを修正できるよう、詳細なドキュメントを収録するために努力が払われました。