Adobe Acrobat®

Adobe® Acrobat®は、PDF（Portable Document Format）ファイルを用いて、
ドキュメントをそのままの体裁で電子的に配信することを可能にしたソフトウェアです。

ご覧のPDFファイルは、用途を配信実験に限定した上で、
開発中のAcrobat Reader 3.0Jを用いて作成したものです。
制作者とアドビシステムズ株式会社両者の事前の承諾なしに、
再配布ならびに、電子掲示板、CD-ROM等への転載、およびデータの改変、
再利用をすることはできません。

Adobe、Adobeロゴ、Adobe AcrobatはAdobe Systems Incorporated（アドビ システムズ社）の商標です。

Newton® Technology
JOURNAL
Volume II, Number 2
April 1996

目　次

Newton Directions
C++とNewtonScript 1

New Technology
アップル社の新製品発表：
Newton OS v2.0搭載のMessagePad 130 1

Communications Technology
有限状態マシン：
Newton通信用モデル 3

Technology Update
パッケージについて 6

New Technology
パッケージの抑制とフリーズ
（Newtonの“シフトキー”） 10

Communications Technology
情報を移動してみよう！ 12

Communications Technology
トランスポートを書く 17

Business Opportunities
Newtonで決め手の
アプリケーションを創造する
七つの方法 22

Letter From the Editors
Newtonデベロッパの皆様へ 23

Newton Directions

C++とNewtonScript

Walter Smith and Rick Fleischman, Apple Computer, Inc.

Newton C++ Toolsが1996年後半にリリースされると、Newtonソフトウェアデベロッパは、C++コードをNewtonのアプリケーションに組み込むことができるようになる。しかし、Newtonソフトウェアのアーキテクチャーはやや独自なものであって、Newtonプラットフォーム上でのC++の使用法は、他のプラットフォーム上での使用法とは異なる。NewtonScriptは、今まで通り高レベルのアプリケーション開発用言語であるのに対し、C++は、必要に応じパフォーマンスを最適化したり、すでにC又はC++で書かれたコードを再利用したい場合に使用される。

本稿では、C++とNewtonScriptの関係について述べる。まずこれらの言語自体について述べ、つぎに、これらの言語の設計の背後にどのような意図があるのかに触れる。さらに、これら二つの言語をニュートン環境で組み合わせる場合の、実際的側面についても述べることにする。

二つの言語のちがい

NewtonScriptとC++の設計は、大きく異なる目的と優先順位のもとでおこなわれた。結果として、これら二つの言語が違ったものになったのは、驚くべきことではない。

C++

C++の目的は“より良いC”をつくりだすことであった。C言語を引き継ぎ、クラスなど大規模プログラミングに適した機能性を持たせることであった。C++のもっとも重要な設計目的のいくつかを列挙すると、以下の通りである。

- Cとの互換性
- Cに匹敵する速度とコードのスリム化
- 既存の開発システムとの統合

C++は、Cの上位集合となることを目的として設計されている。これがなによりも重要であると考えられている理由は、Cが最も人気のあるシステムプログラミング言語だからである。C++はCの低レベル動作を含むので、ハ

20ページへ続く

New Technology

Newton OS 2.0搭載の新機種MessagePad 130をApple社が発表！

by Korey McCormack, Apple Computer, Inc.

Apple Computer Inc.は、MessagePadファミリー製品の最新機種であるMessagePad 130を発表した。Best of Comdex賞に輝くNewton 2.0オペレーティングシステム搭載のMessagePad 130は、モバイルなプロフェッショナルむけの最初のNewton PDAであり、ユーザー制御のバックライトをオンデマンドで使用でき、どのような照明条件の下でも情報の確認と入力を行えるノングレアスクリーンをそなえている。MessagePad 130は、Internet 通信アプリケーションとマルチタスキングのパフォーマンス向上を目的として、システムメモリを増設している。

Newton 2.0の実力

Newton 2.0搭載のMessagePad 130は、ユーザーの仕事と個人生活のオーガナイズを助け、パーソナルコンピュータとの間で情報をシームレスに統合し、ファックス・無線ページング・e-mailなどの通信機能を持ち、さらに広範囲のサードパーティーソリューションにより機能を拡張することができる。

手書き文字認識能力が向上し、データ入力、活字体及び筆記体の認識、筆記体と活字体を混在させてのテキスト入力が容易になった。MessagePad 130には、ノートパッド、To do リスト、スケジュール機能、電話帳、アドレスブックが組み込まれ、プライベートな情報やビジネスの情報などを管理できる。

新しいLCD

MessagePad 130は、バックライトをそなえ、どのような照明条件の下でも情報の確認ができる。薄暗い会議室でも、暗い倉庫の中でも、夜間の自動車内でも、オンデマンドでバックライトを使用でき、情報を見たりアクセスし

21ページへ続く

Newton Technology JOURNAL

Volume II, Number 2 1996年4月

発行／Apple Computer, Inc.
マネージング・エディタ／Lee DePalma Dorsey
テクニカル担当、コーディネーティング・エディタ／Gerry Kane
DTSおよびトレーニング担当、コーディネーティング・エディタ／Gabriel Acosta-Lopez
Newtonデベロッパ・リレーションズ、マネージャ／Philip Ivanier
テクニカル・ピア・レビュー・ボード／J. Christopher Bell, Bob Ebert, Jim Schram, Maurice Sharp, Bruce Thompson
寄稿者
J. Christopher Bell, Michael S. Engber, Rick Fleischman, Guy Kawasaki, Korey McCormack, Jim Schram, Walter Smith, Bruce Thompson

制作／Xplain Corporation
発行者／Neil Ticktin
編集助手／John Kawakami
編集助手／Matt Neuburg
アートディレクタ／Judith Chaplin

©1996 Apple Computer, Inc. 1 Infinite Loop, Cupertino, CA 95014, 408-996-1010. All rights reserved.

Apple、Appleロゴ、AppleDesign、AppleLink、AppleShare、Apple SuperDrive、AppleTalk、HyperCard、LaserWriter、Light Bulb Logo、Mac、MacApp、Macintosh、Macintosh Quadra、MPW、Newton Toolkit、NewtonScript、Performa、Quick Time、StyleWriterおよびWorldScriptは、米国その他の国で登録されたアップルコンピュータ社の商標です。

AOCE、AppleScript、AppleSearch、ColorSync、develop、eWorld、Finder、OpenDoc、Power Macintosh、QuickDraw、SNA類s, StarCore、およびSound Managerは米国アップルコンピュータ社の商標です。ACOTはアップルコンピュータ社のサービスマークです。

DDJ (DeveloperDepot Japan) はXplain Corporationの商標です。

MotorolaおよびMarcoはMotorola, Inc.の商標です。

NuBusはTexas Instrumentsの商標です。

PowerPCはInternational Business Machines Corporationの商標であり、所定のライセンス契約のもとで使用しているものです。

WindowsはMicrosoft Corporationの商標であり、SoftWindowsはMicrosoft CorporationのInsigniaによるライセンスのもとで使用している商標です。

UNIXはUNIX System Laboratories, Inc.の商標です。

CompuServe、Pocket QuickenはIntuit、CIS RetrieverはBlackLabs、PowerFormsはSestra, Inc.、ACT!はSymantec、Berlitzの各商標であり、その他の商標はそれぞれの法的所有権者に帰属します。

この出版物に記載の製品名は参照を目的としたものであり、それらの製品の使用を支持あるいは推奨するものではありません。製品の仕様および説明はすべて各メーカーまたはサプライヤーから提供されたものです。アップル社はこの出版物に記載した製品の選択、性能あるいは使用につき一切の責任を負いません。合意、契約、保証はすべてメーカーと将来のユーザの間で直接おこなわれるものとします。

責任の制限
アップル社は、この出版物に記載した製品の内容、ならびにこの出版物の完全性および正確性に関し、一切の保証をいたしません。アップル社は、商品性および特定の目的に対する適合性を含む保証は、明示的・黙示的にかかわらず、一切いたしません。

デベロッパの皆様へのご挨拶

Newton WWセールス＆マーケティング部門からデベロッパの皆様へのご挨拶

Newtonデベロッパの皆様

Newtonシステムズ・グループは、発足以来最高の成果をおさめて、この四半期を終えることができました。Newton 2.0がBest-of-Comdexの栄誉に輝いて以来、好意的なマスコミの報道がどっと流されました。皆様が既に認めていらっしゃる真価を証明するかのように、Newton 2.0の販売台数は見込数の150%に達し、Newtonがヒット商品であることを世界に告げたのです。

今後の私たちグループの目標は、Newton 2.0の発売を契機に高まりつつある認識と好意とを土台として、ソリューション開発にエネルギーを集中させていくことです。私たちが共に成功をおさめる鍵は、アプリケーションの領域を支配することにあります。

Newtonデバイスで利用可能なアプリケーションに魅力を感じた顧客が、統合化、通信、組織化などの機能のみならず、さらに多くの機能を求めてMessagePadを購入しているわけです。サードパーティ製アプリケーションの多くは、デスクトップ上の作業に対応しつつ、オフィスから離れた場所において本当にモバイルなデバイスで作業するという、完全なモバイル・ソリューションを顧客に提供しています。

先日開催されたSI/VARデベロッパ・コンファレンスでは、400以上のデベロッパ（当社の他の顧客ベース）を前にして、このプラットフォームのためのバーティカル（企業向け）、ホリゾンタル（一般ユーザー向け）、および周辺ソリューションのデモンストレーションが次から次へと展開されました。これらのソリューションがまもなくビジネスの成功に必要な臨界量に達することは、出席した誰の目にも明らかでした。閉会の挨拶では、Jim Buckley（Americas社長）、Dave Nagel（研究開発担当上級副社長）、Mike Markulaといったエクゼクティブが、NewtonはApple戦略の中心とはいえないにしても、その未来は明るい、と口をそろえて語りました。

私どもは今後も新機軸を打ちだし、デベロッパの皆様が最高のアプリケーションのアイディアをExtras Drawerで実現できるような、システムの利便性およびツールを開発していく所存です。また、Newtonプラットフォームのマーケティングおよび宣伝活動を続け、ライセンシーを増やすことによりプラットフォームの利用の拡大を促進していきます。

皆様方には、Newtonのすばらしさという私たち共通の思いを今後も深く心に留めていただき、先ずこのプラットフォームでのアプリケーションの実現に最善を尽くしていただくことによって、これまで同様、このNewtonのすばらしさを広く世界に伝え続けていただきたいと存じます。

WW セールス＆マーケティング部門
ディレクター
Mike Lundgren

Communications Technology

有限状態マシン：Newton通信用モデル

by Jim Schram and Bruce Thompson, Apple Computer, Inc.

Newtonと他のデバイスとの通信は複雑なタスクになりがちである。エンドポイント管理から、ユーザーとのやりとりの間には、多くのことが行われている。有限状態マシンを使用して通信をモデル化すれば、通信をベースにしたアプリケーションの設計業務を単純化できる。

有限状態マシンとはなにか？

有限状態マシン（FSMまたは単にステートマシン）は、ステート（state）、イベント（event）、アクション（action）、およびステート間の遷移（transition）の集まりである。図1は単純なFSMの例である。

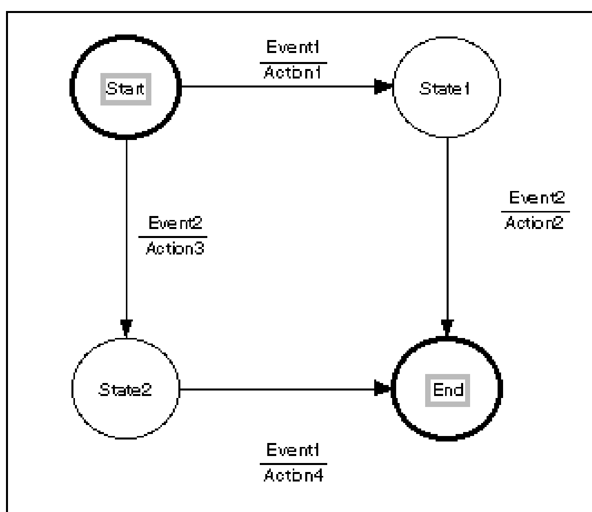


図1. 単純な有限状態マシン

このFSMには4つのステート（スタート、ステート1、ステート2、およびエンド（end））があり、2つの異なるイベント（イベント1、イベント2）に対応している。アクション（action）は、1つのステートから次のステートへ移るときに起こる。たとえば、FSMがスタート・ステートにあるときにイベント1が起こると、FSMがステート1へ移ったときにアクション1が実行される。アクションの意味は、言葉（あるいはコード）で説明するより、（図1のような）ステート遷移図を書いて示した方がわかりやすい。

ステート遷移図があれば、次のような遷移表を簡単に作成できる。

	イベント	*イベント?*
カレント・ステート	イベント1	イベント2
スタート	（アクション1、ステート1）	（アクション3、ステート2）
ステート1	–	（アクション2、エンド）
ステート2	（アクション4、エンド）	–
エンド	–	–

ステート遷移表は、Newton通信用のFSMを作る手がかりになる。ステート遷移表、NTK、およびprotoFSM（以下を参照のこと）を利用すれば、有限状態マシンは非常に簡単に構築できる。

protoFSM

protoFSM、protoState、およびprotoEventは、有限状態マシンをたやすく構築するために利用できるユーザープロトタイプの集合体である。ステートマシン、ステート、およびイベントの関係は、View（ビュー）を作成する時の関係に似ている。ステートマシンが親とすれば、ステートは子である。各ステートは、そのステートに対応する各イベントを子として含んでいる。イベントは、アクション関数と、そのアクションが完了した後の遷移先である次ステートのシンボルを含んでいる。

ステートマシン自身は少数の付加スロットを持つ。varsスロットは、アクションが使う可能性のある付加変数を含むフレームである。varsフレームに入るものの例としては、まず第一にエンドポイントが挙げられる。通信ベースのステートマシンにおいては、アクション・プロシージャの多くがエンドポイントにアクセスしなければならないためである。この他にも、カレント・ステート、カレント・イベント、およびカレント・アクション・プロシージャを反映するスロットがある。[1]

ステートマシンは、いったんセットアップしてしまえば、単にマシンのdoEventメソッドを呼び出すだけで使用できる。doEventは2個のパラメータをとる。第1のパラメータはイベントのシンボルで、第2のパラメータはアクション・プロシージャの付加パラメータの配列である。アクション・プロシージャは、カレントのステート、イベント、および付加パラメータが渡されると、ステートマシンのコンテキスト（注：式やメソッドを評価するときのフレーム＝コンテキストフレーム）で呼び出される。アクション・プロシージャが戻ると、ステートは、イベントに対して定義されたnextStateに従って変化する。

なぜステートマシンを使うのか？

Newton通信アプリケーションの多くは、本質的に並行して動作する2つのメイン“タスク”（task）2を持つ。2つのメインタスクとは、ユーザーインタフェース管理（一般的には、これがアプリケーションの主要動作）と通信管理である。この区分けの例としては、Newton DTSから入手できるLlama-Talkサンプルがあるが、このアプリケーションは、ADSP接続を通じ多様な種類のオブジェクトを送るユーザーインタフェース・エレメント（ボタン）を持つ。ユーザーインタフェース・エレメントが、オブジェクト送信要求を待機中の処理列（キュー。以下キューとする。）に入れると、アイドルスクリプトが実際の通信を実行する。

ステートマシンの動作も同様である。通常、ユーザーインタフェース・エレメントは、ユーザーの要求に基づき、イベントをステートマシンに通知する。ユーザーの要求には、接続の初期化、切断、アイテム送信などのオペレーションなどが含まれる。イベントへの応答（アクション・プロシージャ）は、実際のエンドポイント呼出しを非同期的に実行し、完了スクリプトはアクションの成功または失敗を示すイベントを通知する。

ステートマシンの一例

以上をわかりやすく説明するために、単純なエンドポイントのセットアップとティアダウン（切り離し）を行うステートマシンを例にとろう。このエンドポイントは、シリアル接続を確立し、単純アイテムの送受信を実行し、ユーザーアクションに応じて（あるいはエラー発

生時に）切断を行う。まず最初に、このエンドポイント・ステートマシンの挙動を示すステート図（図2参照のこと）を作成する必要がある。

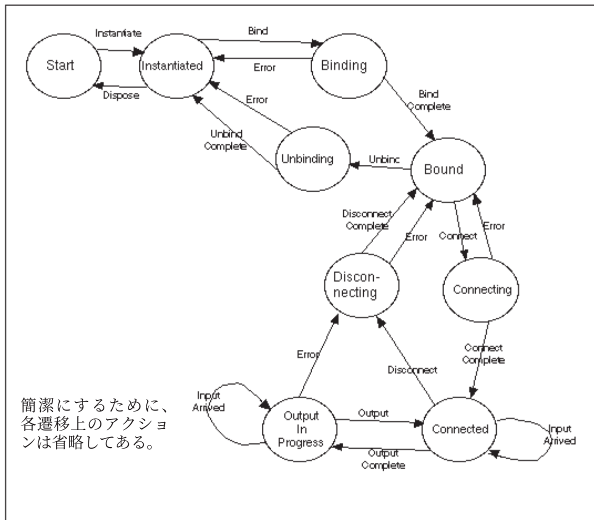


図2：ステート図

次に、この図からステート遷移表を作成する。見やすくするためにアウトラインのみを示した。この表により、ステートマシンがNTKにおいてどのような形をとるか、おおよそ理解できるだろう。

- スタート・ステート (Start State)：
  - インスタンス化イベント (Instantiate Event)
    - 次のステート (NextState)：インスタンス化完了 (Instantiated)
    - アクション：varsにエンドポイントフレームを作成し、ep: Instantiate( )を呼び出す。

- インスタンス化完了ステート (Instantiated State)
  - 結合イベント (Bind Event)：
    - 次のステート (NextState)：結合 (Binding)
    - アクション：ep: Bindを非同期的に呼び出すcompletionScriptがErrorまたはBindCompleteを通知する。
  - 廃棄イベント (Dispose Event)：
    - 次のステート (NextState)：スタート (Start)
    - アクション：ep: Dispose ( )を呼び出し、エンドポイントフレームを廃棄する。

- 結合ステート (Binding State)：
  - BindCompleteイベント (BindComplete Event)：
    - 次のステート (NextState)：結合完了 (Bound)
    - アクション：なし
  - エラーイベント (Error Event)：
    - 次のステート (NextState)：インスタンス化完了 (Instantiated)
    - アクション：エラーが起きたというNotifyを通知する。ここからDisposeイベントを通知することもできる！。

- 結合完了ステート (Bound State)：
  - 接続イベント (Connect Event)：
    - 次のステート (NextState)：接続 (Connecting)
    - アクション：ep:Connectを非同期的に呼び出す。completionScriptがErrorまたはConnectCompleteを通知する。
  - 結合解除イベント (Unbind Event)：
    - 次のステート (NextState)：結合解除 (Unbinding)
    - アクション：ep:Unbindを非同期的に呼び出す。completionScriptがErrorまたはUnbindCompleteを通知する。

- 接続ステート (Connecting State)：
  - ConnectCompleteイベント (ConnectComplete Event)：
    - 次のステート (NextState)：接続完了 (Connected)
    - アクション：inputSpecをセットアップする。InputScriptがInputArrivedイベントを通知し、エラーが起きた場合は、completionScriptがInputErrorを通知する。
  - エラーイベント (Error Event)：
    - 次のステート (NextState)：結合完了 (Bound)
    - アクション：エラーが起きたというNotifyを通知する。ここからUnbindイベントを通知することもできる！。

- 接続完了ステート (Connected State)：
  - InputArrivedイベント (InputArrived Event) ：
    - 次のステート (NextState)：接続完了 (Connected)
    - アクション：入力を処理する。
  - 出力イベント (Output Event)：
    - 次のステート (NextState)：OutputInProgress
    - アクション：ep:Outputを非同期的に呼び出す。completionScriptがErrorまたはOutputCompleteを通知する。
  - 切断イベント (Disconnected Event)：
    - 次のステート (NextState)：切断 (Disconnecting)
    - アクション：キャンセル・オプションを選択すると、ep:Disconnectを呼び出す。completionScriptがErrorまたはDisconnectCompleteを通知する。

- OutputInProgressステート(OutputInProgress State)：
  - OutputCompleteイベント (OutputComplete Event)：
    - 次のステート(NextState)：接続完了 (Connected)
    - アクション：なし
  - InputArrivedイベント (InputArrived Event)：
    - 次のステート (NextState)：OutputInProgress
    - アクション：入力を処理する。
  - エラーイベント (Error Event)：
    - 次のステート (NextState)：切断 (Disconnecting)
    - アクション：エラーが起きたというNotifyを通知する。cancelオプションを選択すると、ep:Disconnectを呼び出す。completionScriptがErrorまたはDisconnectCompleteを通知する。

- 切断ステート (Disconnecting State)：
  - DisconnectCompleteイベント (DisconnectComplete Event)：
    - 次のステート (NextState)：結合完了 (Bound)
    - アクション：なし
  - エラーイベント (Error Event)：
    - 次のステート (NextState)：結合完了 (Bound)
    - アクション：エラーが起きたというNotifyを通知する。ここからUnbindイベントを通知することもできる！。

- 結合解除ステート (Unbinding State)：
  - Unbind Completeイベント (UnbindComplete Event)：
    - 次のステート (NextState)：インスタンス化完了 (Instantiated)
    - アクション：なし
  - エラーイベント (Error Event)：
    - 次のステート (NextState)：インスタンス化完了 (Instantiated)
    - アクション：エラーが起きたというNotifyを通知する。ここからDisposeイベントを通知することもできる！。

上記の表を使えば、NTKにおけるステートマシンのエレメントを簡単にレイアウトできる。最後に知っておくべき点は、パラメータをイベントと一緒にアクション・プロシージャへ渡せることである。たとえば、Outputイベントは、出力すべきアイテムなどの付加パラメータをとることができる。エラーイベントは、例外フレームなどの付加パラメータをとることができる。一般的に言って、アクションはなるべく単純化し、ステートの中に挙動をなるべく多く取り込むのが一番良いやり方である。

まとめ

有限状態マシンは、アプリケーションの挙動をモデル化するための単純で洗練された方法と言える。ステート図を参照すれば、すべての偶発例が処理されているかどうか、簡単に確認できる。protoFSMは、Newtonアプリケーション用のステートマシンを指定する洗練された方法を提供する。通信プロトコルは、有限状態マシンにより定義されることが多い。通信プロトコル定義を有限状態マシンに簡単に転記することができれば、開発時間を削減し、通信コードの複雑さを緩和できるだろう。

Newtonプラットフォームにおける通信は複雑であるが、ステートマシンを利用すれば、その複雑さをもっと扱いやすいレベルに引き下げられる。ステートマシンは、非同期通信の実行により適しているため（同期通信呼出しは相当量のオーバーヘッドを伴う）、これを使用してアプリケーションのパフォーマンス向上をはかることができる。

非同期通信を使用するアプリケーションは、一般的に、同期通信を使用するアプリケーションよりも効率的である。なぜならば、同期通信の呼出しは相当量のオーバーヘッドを伴うためである。ただし、非同期通信アプリケーションの方がプログラミングが難しい場合もある。しかし、有限状態マシンは、複雑さを低減することにより、非同期通信のプログラミングを単純化できるため、結果的にはパフォーマンスの優れたアプリケーションが得られる可能性がある。

1 protoFSMはNewton DTSのサンプルであり、（http://dev.info.apple.com/newton/newtondev.html）で検索できる。

2 この"タスク”という語は、マルチタスキングのような意味合いに取り違えないでいただきたい。Newton 2.0 OSはマルチタスキング（もっと正確に言えば、マルチプログラミング）のオペレーティングシステムだが、NewtonScriptをベースにしたアプリケーションにはマルチタスキングは使えない。

NTJ

AppleのNewton デベロッパプログラムに関するお問い合わせ、
またはお申し込みは下記へご連絡ください。

アップルコンピュータ株式会社
Newton Developer Support事務局
電　話：03-5334-2480
ＦＡＸ：03-5334-2781
email: jnewtondev@asia.apple.com

Technology Update

パッケージについて

by Michael S. Engber, Apple Computer, Inc.

この記事では、Newton OS 2.0のパッケージの変更点をいくつか取り上げる。パッケージ・フォーマット、Unit Import/Export、および新しいExtras Drawer（引きだし）フックの詳細などの関連事項については、パッケージとパートに対する関係を指摘する程度にとどめておく。これらのことがらに詳しく言及するのは、この記事の範囲を超えるためである。

この記事の情報を役立てるためには、Newton OS 2.0用のアプリケーションを書くためにNewton Toolkit (NTK)を使用した経験があり、パッケージおよびパートの基本概念を知っていることが必要である。また“Bit Parts”という記事（本記事末尾の“参考文献と推奨文献”の項を参照のこと）を読んでおくとよいだろう。

注意すべき点として、この記事で取り上げるNTKのVersion1.6のみがサポートしている場合があることに留意する。V1.6以前のNTKはパートをサポートしない可能性があり、将来のバージョンは、より便利な方法でサポートする可能性がある。

基本事項

たいていの人はパッケージとパートの概念を理解していると思い込んでいるようだが、“オートパッケージ”とか“私のパッケージのRemoveScript”など、意味をなさない表現を耳にする機会が多い。パッケージとパートは混同しやすいのである。単に言い間違えに過ぎないこともあろうし、あるいは重大な誤解を示している場合もあろう。

パッケージは、ゼロあるいはゼロよりも多いパートの集まりである。（パートを含まないパッケージも確かに可能ではあるが、無意味である。）小さなパッケージの例を図1に示す（16進）。

```
7061636B616765306E6F6E6500000000
00000001000000020000000200000036
00000000000000000000000000000036
000000000058
```

図1：世界最小のパッケージ（54バイト）

通常のパッケージは54バイトよりいくぶん大きく、1個または1個を超えるパートから成る。表1は、一般的なパートの種類を列挙したものである。

タイプ	フレームパートか？	説　明
form	Yes	アプリケーション
book	Yes	ブック
auto	Yes	テクニカルハッキング
font	Yes	追加フォント
dict	Yes	カスタム・ディクショナリ
soup	No	読み取り専用スープによるストア

表1：パートのタイプ

表1の第2列は、パートのタイプがフレームパートであるか否かを示している。フレームパートは、基本的にはただ1つのNewtonScriptオブジェクト、すなわち1つのフレームに過ぎない。フレームに何が入るかは、パートのタイプによって異なる。たとえば、フォームパートは、the Formという名のスロットにそのベースビューのテンプレートを保持する。

GetPkgRefInfoによって戻されるフレームのpartsスロット経由で、（パッケージのフレームパートの）実パートフレームにアクセスすることができる。この点に関しては、わずかながら誤解がある。フォームパートのInstallScriptへ渡される引き数は一般的にpartFrameと呼ばれるが、実際はパートフレームではない。これは、1.xバージョンにおいてはパートフレームのクローンであり、2.0バージョンにおいては、パートフレームのプロトであるフレームである。ただし、この点は変更される可能性がある。したがって、partFrame.slotは、パートフレームから指定されたスロットを戻すという事実だけを覚えておけばよい。

パートのタイプはほとんどがフレームパートである（ストアパートは最も注意すべき例外である）。非フレームパートのタイプ（例：通信用ドライバ）を作成することは可能だが、そのためのツールがまだ用意されていない。

パッケージ・オブジェクト

Newton OS 1.xにおいては、パッケージに直接アクセスする方法がなかった。 GetPackages呼出しはアクティブ・パッケージに関する情報の配列を戻したが、パッケージ自身であるオブジェクトは戻されなかった。

Newton OS 2.0では、パッケージが仮想バイナリ・オブジェクト（VBO）としてストアされる。パッケージVBOへの参照を、パッケージ参照（関連する関数名においてはpkgRefと略されることが多い）と言う。 多くの場合、パッケージVBOは、store:NewVBOで作成する通常のVBOと同じである。たとえば、GetVBOStoreは、パッケージが常駐するストアを決定するのに使用できる。

パッケージをこのような方法で表現することのデメリットは、通常のVBOと同様、パッケージVBOはスープの中に格納されて、ハッキングの格好の標的になりやすいという点である。このスープの詳細は秘密で、変更を受けやすい。 サポートされるオペレーション用にはAPIが用意されている。サポートされるAPIの範囲外でデベロッパが行うことはすべて、（マイナーではあるが）現行のシステムに対して干渉する可能性が高い。将来のシステムの損傷を招く可能性があることは言うまでもない。

パッケージ参照と通常のVBO

多くの使用目的において、パッケージVBOは、NTKで作成したパッケージファイルからのバイナリデータを含むものと考えてよい。パッケージのロード時に満たされる少数のプレースホルダ・フィールド（ローディングの時間など）があるが、これらはバイトの面から見ると等しい。あるいはそう見える。

VBOを作成する場合は、そのクラスを'packageに設定した後、BinaryMungerを用いて、“本物の”パッケージからバイトのコピーを行う。すると、パッケージに似たものを得られるが、これは本物のパッケージではない。それをIsPackageに渡してみよう。

両者の違いはわかりにくい。一つはっきりしているのは、本物のパッケージは読み取り専用だという点である。もう一つの、もっと根本的な違いは、偽のパッケージは、通常のパッケージローディング・プロセス中に、パッケージに対応付けられる再配置情報がない点である。この再配置情報は、バイナリデータに含まれていないため、ExtractByteによってアクセスできない。

偽物から本物のパッケージを得る唯一の方法は、そのパッケージを

store:SuckPackageFromBinaryに渡し、データから本物のパッケージを作成することである。偽物をそのまま本物にうまく変換する方法はない。

Extras Drawer

1.xでは、Extras Drawerはアプリケーションを開始するためにのみ使用した。Extras Drawer内のアイコンは、フォームパートとブックパートに対応していた。パッケージの削除は、ポップアップメニューが、パートではなくてパッケージのリストになっている“Remove Software”ボタンにより行った。

Newton OS 2.0では、“Remove Software”ボタンがない。Extras Drawerには、必要なパッケージがすべて揃っている。したがって、Extras Drawerアイコンの下にあるモデルが以前とは異なっている。たとえば、フォームパートを含まないパッケージも表示用アイコンが必要であり、それによってユーザーが削除を行う対象を指定することができる。

2つのソリューションが考えられる。すなわち、各パッケージに対して1つのアイコンを持つか、あるいは各パートに対して1つのアイコンを持つかである。このどちらをとっても十分とは言えない。次の例を考えてみればわかるだろう。

- ある種のマルチパート・パッケージには、アイコンが2つ必要（例：フォーム+ブックパート）
- ある種のマルチパート・パッケージには、アイコンが1つだけ必要（例：フォーム+オートパート）
- 非フレームパートには、タイトルおよびアイコンのデータを備える場所がない（すなわち、フレームパートがない）

Extras Drawerが採用しているアプローチは、2つのソリューションをかけあわせたものである。これは、パッケージ構築におけるデベロッパの権利保障（Miranda Rule）のようなものである。すなわち“あなたは自分のアイコンを作る権利を持つ。しかし、あなたにアイコンを作る余裕がなければ、あなたのためにアイコンが1つ割り当てられる。”というルールである。

パッケージ内のフレームパートは、そのパートフレーム内にテキストスロット（およびオプションでアイコンスロット）を備えることにより、Extras Drawerアイコンを持つことができる。パッケージ内のフレームパートがアイコンを提供できない場合は、パッケージ名のラベルがついた総称アイコンが作成される。

このアプローチは、現行のパッケージの1.x表現を保存するとともに、将来作成されるもっと複雑なパッケージに対するフレキシビリティを備えている。現行のパッケージの大多数は1つのフォームパートから成っているため、今まで通り、1つのアイコンをとる。マルチパート・パッケージもまた正しく扱われる。フォーム+ブックパートの場合は2つのアイコンをとり、フォーム+オートパートの場合はただ1つのアイコンをとる。

残念ながら、現行のパッケージでは、フォームパートもブックパートも含まないパッケージは“システムが割り当てた”表現をとる。つまり、パッケージ名のラベルがついた1つの総称アイコン（図2を参照のこと）である。パッケージ名以外の情報がないので、これ以上のことはできない。これらのパッケージを再構築して、タイトルとアイコンをもっと見栄えのよいものにしたければ、NTK 1.6を使用していただきたい。

図2：総称アイコン

アイコンを作成できるからといって、パッケージのすべてのパートにアイコンを用意する必要はない。ほとんどのパッケージについては、メインのフォームパートに対して1つのアイコンで十分である。余分なアイコンは役に立たないばかりか、ユーザーをいたずらに混乱させるだけである。

```
SetPartFrameSlot('title, "foo");
SetPartFrameSlot('icon, GetPICTAsBits("foo picture", true));
```

作成するパッケージがフレームパートを含まない場合（たとえば、スープパートしか含まないなど）、総称アイコンをとらないようにするには、タイトルとアイコンを指定するダミーのフレームパートを加える。たとえば、パッケージに関する情報を表示するフォームパートを加えてもよいだろう。

アクティブ・パッケージと非アクティブ・パッケージ

パッケージのInstallScript（フォームパートの場合は、ベースビューの作成）の実行は、パッケージをアクティベートするプロセスのひとつである。RemoveScriptの実行は、パッケージを非アクティベートするプロセスのひとつである。Newton OS 2.0以前においては、パッケージはインストール時にアクティベートされ、削除時に非アクティベートされた。サードパーティ製ユーティリティを使用しない限り、ストア上に存在し、かつ非アクティブなパッケージという概念はなかった。（ここで非アクティブなパッケージとは、InstallScriptがまだ実行されていないか、あるいは、パッケージがあらかじめアクティブになっていれば、非アクティベートのプロセスとしてRemoveScriptが実行されたパッケージを意味する。）

Newton OS 2.0では、アクティブでないパッケージでもExtras Drawerに表示できるようになった。ユーザーが故意にパッケージを“フリーズ”した場合は、スノー・アイコンが表示され、エラーあるいはユーザーがパッケージのアクティベートを抑制したために非アクティブになっている場合は、×印のついたアイコンが表示されて、それぞれ識別できるようになっている。（詳細については、“パッケージの抑制とフリーズ”の記事を参照のこと）

ほとんどの場合、アプリケーションはこのような詳細なことまで意識する必要はない。インストールを正確に処理し、さらに（削除操作または強制的なカードイジェクトによる）削除を正確に処理するパッケージであれば、ユーザーによるアクティベート、非アクティベートと協調して動作するはずである。

無効参照(Invalid Reference)

無効参照は、非アクティブまたは使用不能な（すなわち、削除のプロセスにあるカード上の）パッケージ内にあるNewtonScriptオブジェクトへの参照である。このトピックは、“Newton Still Needs the Card You Removed”という記事の中で扱っている（本記事末尾の“参考文献と推奨文献”の項を参照のこと）。

パッケージ内のオブジェクトへの参照値は、パッケージが非アクティベートされた後は無効になる。1.xでは、そのようなオブジェクトにアクセスしようとすると、-48214エラーが生じた。Newton OS 2.0では、パッケージが非アクティベートされた後は、現在の参照値が特殊値に置き換えられる。つまり、-48214エラー（このエラーは不良なパッケージをロードした時にも生じる）というようなあいまいなメッセージではなく、たとえば次のような、より具体的なエラーメッセージをNTKインスペクタから得られる。

```
!!! Exception: Pointer to deactivated package
```
あるいは
```
<bad pkg ref>
```
あるいは
```
bus error with value 77
```

削除のプロセスにあるカード上のパッケージについても、同様の問題が起こる。パッケージ内のオブジェクトが使用不能であれば、アクセスしようとすると、あの不愉快なカード再挿入ダイアログが現われる。Newton OS 2.0では、カード再挿入ダイアログはなくなってはいないが、多少改善されている。現在では、無効参照を使用したパッケージ名が表示される。この他に、カード再挿入ダイアログが表示される原因には、パッケージをたやすく特定できない、あまり一般的でないものがある（たとえば、ストアへのアクセスなど）。このようなケースにおいては、パッケージ名は表示されない。

トラブルの原因となっているパッケージの名前がわかるということは、カードを取り外せない理由をつきとめようとするユーザーにとってはありがたい。しかし、これは、パッケージ自体が“にっちもさっ

ちも行かない”理由をつきとめようとするデベロッパには、ほとんど役立たない。こういった場合のために、デベロッパ用ツールを提供するプランがいくつかある。そのツールを用いれば、どの無効オブジェクトにアクセスしたか判断できる。

このような問題の処理に利用できる新しい関数もある。以前は、参照が有効であるかどうかは、アクセスしてみなければ判断できなかった。Newton OS 2.0では、IsValid 関数を用いることにより、参照が有効かどうかの判断ができる。

新しいパートフレーム・スロット

Newton OS 2.0には新しいパートフレーム・スロットが用意されている。このスロットのあるものはデータであり、あるものはメソッド（すなわち関数を含む）である。前に述べたように、このスロットに値を設定するには、NTKのSetPartFrameSlot関数を用いる。

新しいパートフレーム・データスロット

`app`
`icon`
`title`

1.xでは、app、icon、およびtitleスロットをそれぞれ用いて、フォームパートのappSymbol、Extras Drawerアイコン、およびExtras Drawerタイトルを指定した。2.0では、これらのスロットはどのフレームパートにも使える。非フォームパートについては、SetExtrasInfo使用時にappSymbolを用いてパートを識別する。

`labels`

このスロットはExtras Drawerアイコンを前もってファイルするのに用いる。たとえば、アイコンが初期状態においてSetUpフォルダに入るよう指定するには、'_SetUpを用いる。詳細な例については“ Extra, Extra ”（本記事末尾の“ 参考文献と推奨文献 ”の項を参照のこと）の記事を見ていただきたい。

新しい（オプショナル）パートフレーム・メソッド

`DoNotInstall`

DoNotInstallメッセージは、パッケージがユニットにインストールされる前に、（引き数なしで）送られる。これにより、パッケージは自身のインストールを止めることができる。このメッセージはパッケージ内のすべてのフレームパートに送られる。その中で非nil値を戻すフレームパートがあれば、パッケージはインストールされない。インストールの失敗についてユーザーに何も知らせないよりも、何らかのフィードバックをした方がよい。たとえば、必ず内部ストア上にのみインストールするパッケージであれば、次のようなDoNotInstallになるだろう。

```
func()
begin
  if GetStores()[0] <> GetVBOStore(ObjectPkgRef('foo)) then
    begin
      GetRoot():Notify(kNotifyAlert, kAppName, "This package
was not installed.
        It can only be installed onto the internal store.");
      true;
    end;
end
```

`DeletionScript`

DeletionScriptメッセージは、パッケージ削除の直前に（引き数なしで）送られる。このスクリプトにより、ユーザーによるパッケージのスクラビング（ジグザグ動作）とユーザーによるパッケージのカードの強制イジェクトを区別できる。DeletionScriptは通常、スープ削除、ローカルフォルダ削除、およびシステムスープからのプレファレンス削除などの“ クリーンアップ ”を行う。

`RemovalApproval`
`ImportDisabled`

これらのスクリプトは、パッケージがインポートしているユニットのいくつかが、取外し中であることをパッケージに通知する。RemovalApprovalにより、パッケージは、ユニット取外し前に、取外しの影響をユーザーに警告することができる。ImportDisabledは、ユニットの取外しが最終的に実行された場合に送られる。ユニットに関する詳細については、DTSサンプルコードの“ MooUnit ”を参照していただきたい。

新しいパッケージ関連関数

`GetPackageNames(store)`

戻り値―パッケージ名の配列（文字列）
store―パッケージが存在するストア

GetPackageNamesは、指定したストア上にあるすべてのパッケージ名を戻す。アクティブでないパッケージ（例：フリーズされたパッケージ）も含めて、すべてのパッケージ名を戻す。

`GetPkgRef(packageName, store)`

戻り値―パッケージ参照値
packageName―パッケージの名前
store―パッケージが存在するストア

GetPkgRefは、パッケージ名とそれが存在するストアを指定すれば、パッケージへの参照値を戻す。

`ObjectPkgRef(object)`

戻り値―パッケージ参照値（オブジェクトがパッケージの中になければnil）
object ―任意のNewtonScriptオブジェクト

オブジェクトがどのパッケージに入っているかを決定し、パッケージ参照値を戻す。スープエントリを含むNewtonScriptヒープ内の即値(immediate)およびその他のオブジェクトに対しては、nilを戻す。

パッケージが自身のパッケージ参照値を得るには、任意の非即値(non-immediate)リテラルでObjectPkgRefを呼び出す。たとえば、ObjectPkgRef('read, ObjectPkgRef("my")、またはObjectPkgRef('[1,i,p,s])とすればよい。フォームパートのInstallScriptはクローン化されており、（EnsureInternalによって）そのクローンが実行されることに注意していただきたい。つまり、コードブロック全体が、ObjectPkgRefへの引き数を含め、NewtonScriptヒープ内に存在するため、上記の例が機能しないのである。この場合の対策は、InstallScriptへ渡される引き数により、パッケージからオブジェクトを得ることである。たとえば、ObjectPkgRef(partFrame.theForm)とすればよい。その他のパートタイプについては、この問題は起こらない。

`GetPkgRefInfo(pkgRef)`

戻り値―infoフレーム（以下を参照のこと）
pkgRef―パッケージ参照値

この関数は、以下に示すように、指定したパッケージに関する情報のフレームを戻す。

```
{
  size:           <int>            // uncompressed package size in bytes,
  store:          <frame>          // store on which package resides
  title:          <string>,        // package name
  version:        <int>,           // version number
  timestamp:      <int>,           // date package was loaded
  creationDate:   <int>,           // date package was created
  dispatchOnly:   <nil or non-nil>, // is package dispatch-only?
  copyprotection: <nil or non-nil>, // is package copyprotected?

  copyright:    <string>,          // copyright string
  compressed:   <nil or non-nil>,  // is package compressed?
  cmprsdSz:     <int>,             // compressed size of package in bytes

  numparts:     <int>,              // #parts in the packages
  parts:        <part data array>,  // part-frames for the frame parts
  partTypes:    <array of symbols>, // part types corresponding to data in parts
  slot

  // other slots are private and undocumented
}
```

```
IsPackageActive(pkgRef)
```
戻り値—nilまたは非nil
pkgRef—パッケージ参照値
IsPackageActiveは、指定したパッケージがアクティブか否かを決定する。

```
IsPackage (object)
```
戻り値—nilまたは非nil
object —任意のNewtonScriptオブジェクト
IsPackageは、指定したオブジェクトがパッケージ参照値であるか否かを決定する。

```
IsValid (object)
```
戻り値—nilまたは非nil
object —任意のNewtonScriptオブジェクト
IsValidは、指定したオブジェクトが既にアクティブではないパッケージ内にあるか（あったか）を検出する。パッケージが、取外しプロセスにあるカード上にあれば、nilを戻し、カード再挿入ダイアログを表示しない。IsValidは、パッケージ内に存在しない（たとえばNewtonScriptヒープ内あるいはROM内に存在する）即値またはオブジェクトに対しては、trueを戻す。IsValidはオブジェクトを徹底的にチェックするわけではない点に注意すること。

```
MarkPackageBusy(pkgRef, appName, reason)
```
戻り値—指定されない
pkgRef—パッケージ参照値
appName—パッケージを要求する実体を記述する文字列
reason—パッケージを非アクティベートできない理由を記述する文字列
MarkPackageBusyは、指定したパッケージが使用中であるか否かを示す。これは、ユーザーが警告を受け、そのパッケージを非アクティベートするオペレーション（たとえば、パッケージの削除あるいは移動）を中止できるチャンスを与えられることを意味する。appNameおよびreasonは、ユーザーに対して表示するメッセージを生成するのに用いる。

非アクティベートすると不都合が起こるようなパッケージについては、使用中と表示しなければならない。たとえば、ストアパートは致命的なデータを提供している可能性がある。ユーザーはそれでもオペレーションを続行するかもしれないので、こういった偶発事にも適切に対処できるようにしなければならない。パッケージはできるだけ速やかに解放して、ユーザーが不便を感じないようにすること。

パッケージからユニットをインポートしている場合は、そのパッケージに対してMarkPackageBusyを用いる必要はない。ユニットは、この種の問題を処理するために独自のメカニズム（RemovalApprovalなど）をもっている。

```
MarkPackageNotBusy(pkgRef)
```
戻り値—指定されない
pkgRef—パッケージ参照値
MarkPackageNotBusyは、指定したパッケージがもはや使用中でないことを示す。

```
SafeRemovePackage (pkgRef)
```
戻り値—指定されない
pkgRef—パッケージ参照値
SafeRemovePackageは、指定したパッケージを削除する。そのパッケージが使用中であれば、ユーザーは削除オペレーションを中止するチャンスを与えられる。

```
SafeMovePackage (pkgRef, destStore)
```
戻り値—指定されない
pkgRef—パッケージ参照値
destStore—パッケージの移動先ストア
SafeMovePackageは、指定したパッケージを指定したストアへ移動する。そのパッケージが使用中であれば、ユーザーは移動オペレーションを中止するチャンスを与えられる。パッケージを移動するには、それを非アクティベートし、移動した後に再アクティベートしなければならない。

```
SafeFreezePackage (pkgRef)
```
戻り値—指定されない
pkgRef—パッケージ参照値
SafeFreezePackageは、指定したパッケージをフリーズする。そのパッケージが使用中であれば、ユーザーはフリーズオペレーションを中止するチャンスを与えられる。

```
ThawPackage (pkgRef)
```
戻り値—指定されない
pkgRef—パッケージ参照値
ThawPackageは、指定したパッケージを解凍する。

参考文献と推奨文献

Engber, Michael S., "Bit Parts." PIE Developers, 1994年5月号, pp.27—29
この記事は、Newton OS 1.x内のパッケージと、NTKの古いバージョンでのパッケージ作成を扱っている。内容のほとんどは今なお十分に通用する。この記事はさまざまなPIE DTS CDやアーカイブから入手できる。あるいは次のアドレスからftpで入手可能である。
ftp.apple.com/pub/engber/newt/articles/Bitparts.rtf

Engber, Michael S., "MooUnit." PIE DTS Sample Code, 1995年秋
このサンプルコードは、unit import/exportに関するドキュメンテーションと単純な一例を提供している。これはさまざまなPIE DTS CDやアーカイブから入手できる。

Engber, Michael S., "Newton Still Needs the Card You Removed." Double Tap, 1994年5月号, pp.12—18
この記事は、（非アクティブなパッケージ内のオブジェクトへの）無効参照に関して徹底的な検討を行っている。この記事はさまざまなPIE DTS CDやアーカイブから入手できる。あるいは次のアドレスからftpで入手可能である。
ftp.apple.com/pub/engber/newt/articles/NewtonStillNeedsTheCard.rtf

Goodman, Jerry, "Psychic Archaeology." 1980年, Berkley Books
この本は、ほとんど確かな情報がない出所不明の出土物を調査する技術を述べている。

Sharp, Maurice, "Extra Extra." Newton Technology Journal, 1996年2月号

NTJ

パッケージの抑制とフリーズ（Newtonの“シフトキー”）

by Michael S. Engber, Apple Computer, Inc.

この記事は、どのようにパッケージを抑制して、互換性のないアプリケーションを取り扱うか（そのアプリケーションが削除不能であるなどきわめて深刻なシステム上の問題を引き起す場合も含めて）という実際的な観点から、パッケージ活性化の概念を取り扱う。Newton OS 2.0の潜在的な機能であるパッケージ・フリージングにも関連概念として触れる。

パッケージ活性化の抑制

Newton OS 1.xでは、内部ストア上のパッケージが起動時にトラブルを引き起こす場合、内部ストアを完全に消去する（パワースイッチを押してリセットする）しか方法がなかった。同様に、カード上のパッケージがカードのマウントを妨げる場合も、そのカードを消去する（Prefsオープンしてカードを挿入する）しか方法がなかった。

Newton OS 2.0では、パッケージの活性化を抑制できる。これにより、パッケージのコードをまったく実行せずに、ユニットを起動したり、カードを挿入したりできるようになった。これは、問題のあるパッケージが存在するストアを完全に消去しなくても、問題パッケージを消去できるということである（マッキントッシュのユーザーであれば、機能拡張を取り込まないために、シフトキーを押しながら起動する操作を思い起こすかもしれない）。

内部ストア上のパッケージ活性化を抑制するには、ユニットをいったんリセットしてから起動し、スクリーンの左から約6ミリの所をペンで押し下げる。内部ストア上のパッケージを活性化するかどうかを問うメッセージ（図1を参照のこと）が表示されるまで、ペンを押しつけたままにする。メッセージが表示されたら、“No”を選択する。

（注：ペンがプラスチックケースの高くなっている縁に接触していると、左に寄り過ぎなので、うまくいかない。スクリーンとケースの境界にはわずかなデッドエリアがあるためである。図1のメッセージは、スプラッシュスクリーンが消える前に表示される。したがって、スプラッシュスクリーンが消えてしまったら、ペンを縁からもう少し離してやり直さすこと。）

図1：パッケージ活性化ダイアログボックス

Extras Drawerを開くと、×印のついたアイコンがいくつかあるはずである（図2を参照のこと）。これは、そのパッケージがアクティブでないことを示している。このプロシージャはROMに組み込まれたパッケージに対しては作用しない。このように不活性化できるのは、ユーザー自身がロードしたか、あるいは工場でユニットにプレロードされたパッケージのみである。

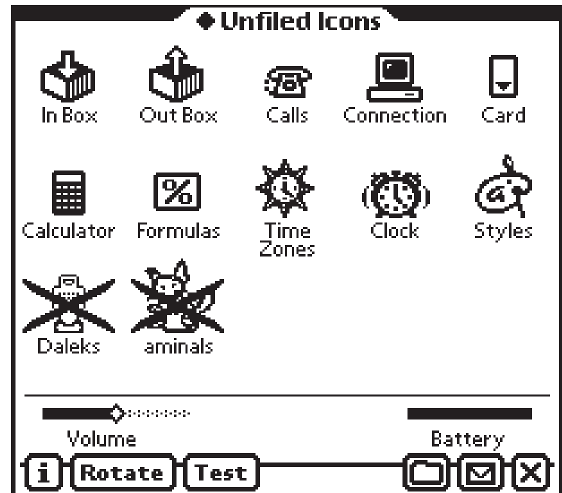


図2：Extras Drawer内の×印をつけられたアイコン

×印のついたアイコンをタップすると、そのパッケージが活性化される。犯人がみつかるまで、一度に1つずつパッケージを活性化させてみよう。たとえば、Namesアプリケーションが開かないのであれば、×印のついたアイコンを押して、×印が消えたらNamesを開けるかどうかを確かめる。

この時点でトラブル原因のアイコンを削除できれば、しめたものである。 Extras Drawerがめちゃめちゃになるくらい問題が厄介であれば、再度リセットしてから、パッケージ活性化を抑制しなければならない。その場合は、最初に活性化しないで単にアイコンを削除すること。

邪魔をしているappsを探す場合は、機能拡張フォルダの中を見てみること（あるいはAll Iconを表示する）。1.xシステムでは、Extras Drawer内に対応するアイコンをもたないパッケージがいくつかあったが、Newton OS 2.0では、どのパッケージも最低1つのアイコンをもっている。

記憶カードの場合も同じ方法が使える。まずカードを挿入する。ロックした後、前に述べたようにスクリーンの左端近くをペンで押し下げる。すると、カード上のパッケージを活性化するかどうかの質問が表示される。

より高度なテクニック

ユニットをリセットし、スクリーンの上端近く（左端ではない）をペンで押し下げると、パッケージ活性化を抑制するのに加えて、ユニットのオリエンテーションおよびバックグラウンド・アプリケーションをデフォルトにリセットできる。すなわちポートレートの書式で、バックグラウンドはNotepadになる。アプリケーションがうまく動いているかにみえたのに、バックグラウンドに切り換えたら問題があったという場合は、このテクニックが役立つ。

上端を押すことがこのような付加効果をもつのは、ユニットをリセットした場合のみである。したがって、記憶カードを挿入する場合は、パッケージの活性化を抑制するのに上端か左端のどちらを押してもよい。

アイコンに×印がつけられるその他の理由

×印のついたアイコンは、パッケージが非アクティブであることを示す。パッケージのローディングを抑制しなくても、非アクティブ・パッケージに遭遇するケースは他にもいくつかある。最も一般的なケースは、同じパッケージの2つのコピーをロードしようとしたときである。たとえば、"foo"パッケージが内部ストア上にあるとき、"foo"を含むカードを挿入すると、カード上の"foo"のアイコンには×印がつけられる。

他にも、ほんの短い間ではあるが、×印を目にするケースがあるかもしれない。たとえば、パッケージをストア間で移動する場合、パッケージは非活性化され、移動後に再活性化される。この場合、パッケージは一時的に非アクティブになるに過ぎないため、×印は表示しないようにする。ただし、場合によってはほんの短い間×印が見えるようにする。

非アクティブのアイコンが表示されていたら、かまわずにそれをタップしてよい。すると、システムがそれを活性化し、起動しようとする。問題があれば、エラーメッセージが表示される。運が良ければ、"（my cardストア上の）'foo'パッケージは、（internalストア上の）同じ名前のパッケージが既に使用中であるため、活性化できませんでした。"などの情報を含むメッセージが表示される。

パッケージ・フリージング

フリーズされたパッケージがどのようなものかご存知でない読者は、この先を読んでいただきたい（スーパーマーケットで売っている冷凍食品ではない）。Extras Drawerのこの潜在的機能を有効にするユーティリティは、多くのサードパーティ・デベロッパが提供している。

フリーズされたパッケージは非アクティブなパッケージである。フリーズされたパッケージと抑制されたパッケージの違いは、前にも述べた通り、フリーズされたパッケージは、ユーザーが故意に非アクティブにしたものであり、ユーザーが再活性化するまで非アクティブな状態にとどまるという点である。パッケージの活性化を抑制するということは、一時的にパッケージを非活性化することである。したがって、ユニットをリセット（あるいはカードを再挿入）すれば、アイコンの×印は消えてしまう。一方、フリーズされたパッケージは起動時にも非活性化されており、ユーザーが指定しない限り、そのままの状態を保持する。

パッケージの活性化は、システムのワークメモリを一部使い果たすし、また時間もかかる。多くのパッケージを含むカードをマウントするのは時間がかかるし、少ないメモリでオペレーションをすることになる。これをうまく切り抜ける方法は、あまり使わないパッケージをフリーズ（非活性化）することである。パッケージまたはパッケージ・グループをフリーズするには、Extras Drawerからフリーズするものを選択し、ActionボタンのFreezeを選択する。すると、パッケージのアイコンがスノー・アイコン（図3を参照のこと）に変わり、そのパッケージがフリーズされたことを示す。フリーズされたパッケージを解凍する（活性化する）には、このアイコンをタップするだけでよい。

図3. フリーズされたパッケージを示すスノーアイコン

NTJ

AppleのNewton デベロッパプログラムに関するお問い合わせ、
またはお申し込みは下記へご連絡ください。

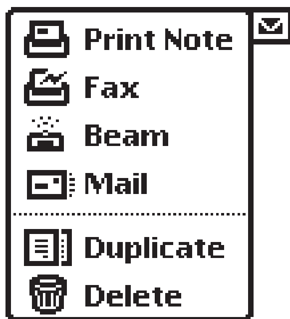
アップルコンピュータ株式会社
Newton Developer Support事務局
電　話：03-5334-2480
ＦＡＸ：03-5334-2781
email: jnewtondev@asia.apple.com

Communications Technology

情報を移動してみよう！

by J. Christopher Bell, Apple Computer, Inc.

モバイルデバイスであるNewtonにユーザーが初めてアクセスするとき、その最大の関心事は情報の入力（あるいは、場合によっては情報の取り込み）であろう。情報を入力して初めて、それをNewtonの外部に、プリンタかもしれないし友人のemailアカウントかもしれないが、移動できるのである。情報をNewtonの外部へ移動する一あるいは削除、コピーする一には、NewtonユーザーインタフェースではおなじみのActionボタン（封筒アイコン）をタップすればよい。Actionボタンをタップすると、現在見えている情報を移動する方法のリストが表示される。



Actionボタンを使って情報を移動する機能を、ルーティングと呼ぶ。

Actionリスト内の点線より上のアイテムは、情報をNewtonデバイスの外部へ移動するサービスである。これらの通信サービスをトランスポートと呼ぶ。Newton OSには数種類のトランスポート機能が組み込まれているが、ユーザーがパッケージとしてインストールできるような、新たなトランスポートをデベロッパが作成してActionリストに選択肢として追加してもよい。

点線より下のアイテムはアプリケーションに固有のアクションで、多くの場合DuplicateとDeleteを含んでいる。どのアプリケーションも、固有のActionリストに、ルートスクリプトと呼ばれるこれらのアクションを追加することができる。アプリケーションは、どのトランスポートをリストの一番上に表示するか制御することができる。

アプリケーションは、サポートするトランスポートを名前でリストするだけではトランスポートを機能させることができない。仮にできたとしても、アプリケーションのデベロッパは、他のデベロッパが新たなトランスポートを作成するたびに、自社製品を新たなトランスポートリストでアップグレードしなければならなくなる。その代りに、アプリケーションは、一般的な言い方で、処理する情報の特性に基づき、どのタイプのトランスポートが適切かを指定するのである。dataTypeは、情報をNewtonデバイスの外部へ送信する方法の記述に用いる、全体クラス（generic classification）である。一般的なdataTypeは、普通テキスト（Mailが使用する'text dataType)、NewtonScriptフレーム（Beamが使用する'frame dataType)、およびイメージング・レイアウト（PrintとFaxが使用する'view dataType）である。独自のdataTypeを定義することもできるが、これを利用するためには、アプリケーションにそのdataTypeの詳細を知らせなければならない。

トランスポートのカテゴリを指定するのにdataTypeを用いるメリットは、たくさんある。最大のメリットは、Newtonプラットフォームによるソリューションにとって、サードパーティのトランスポートが組込みトランスポートと同じくらい重要な役割を担いうるという点である。これは、少なくとも1つの標準的dataTypeをサポートするトランスポートならば、自動的に、ほとんどのアプリケーションと協調して動作するためである。場合によっては、社内デベロッパあるいはバーティカル・デベロッパがカスタム・トランスポートを書くこともできる。アプリケーションとトランスポートは、一緒に働くためにお互いの詳細を知る必要はないのである。

ActionボタンがActionリストを表示させるのは、ユーザーが、情報を移動するためにトランスポートとルートスクリプトを選択する前でなければならない。アプリケーション内にどのアクションが現われるのかを決定するのは、ユーザーがActionボタンをタップしたときであり、アプリケーションがインストールされたときではない。このようなインタフェースには興味深い利点がいくつかある。たとえば、トランスポートは、動的に登録と登録取消しが可能でありながら、しかも一貫したユーザーインタフェースを維持する。このプロセスの動的な特性ゆえに、ルーティングコードを書く場所あるいは書き方に関して、フレキシビリティが得られるのである。したがって、一定のメソッドとスロットを提供し、グローバル・レジストリを用いることにより、システムがActionリストをビルドするのを助けていかなければならない。ルーティングを実行するということは、Actionボタンを使用し、送受信箱（In/Out Box）で通信するためのコードを書くことなのである。

ルーティングコードを書く前に

ルーティングをサポートするアプリケーションのコーディングにとりかかる前に、考えておくべきことがいくつかある。ルーティングを書く前に答を出しておかねばならない最も重要な問いは、“このアプリケーションには、何種類のデータがあるのか？”ということである。各データタイプは、デベロッパ・シグネチャを含む一意的なシンボルにより識別しなければならない。たとえば、開発中のアプリケーションが2つのメインビューをもち、そのうち1つはラマ（注：Llama（ラマ＝動物）とは、Apple社のPersonal Interactive Electronics(PIE)部門のDeveloper Technical Supportグループの非公式マスコットです。）についての情報を、もう1つはラマ牧場主（＝人間）の情報を処理するとすれば、データを表現するには、2つのデータタイプを用いても良い。データのタイプ一これをデータクラスと呼ぶ一は、`'|llama:jX|`と`'|rancher:jX|`のようなシンボルをもっても良い。

データクラスの数を決定するためには、データのルーティングが可能なトランスポートのタイプを考えるとわかりやすい。たとえば、先の例でいえば、牧場主情報はテキストオンリーのe-mailシステムへエクスポートできるが、ラマの情報は画像のみで構成されるかもしれないため、テキストオンリーのトランスポートへ送っても意味がない。つまり、情報は複数のデータクラスで構成されるため、それらのデータクラスが何を表現するのかを理解しておかねばならない。

ユーザーがアプリケーションから情報を印刷するとき、印刷レイアウト（これを印刷フォーマットと呼ぶ）は、アプリケーションの特定のパートからのみアクセス可能だろうか？もしそうならば、アプリケーションのサブビューは異なるデータクラスを表現する可能性があり、これらのデータクラスが何を表現するのか理解しておかねばならない。

ルーティングコードを書く前に、ユーザーがActionボタンをタップしたときにアプリケーションまたはそのサブビューが処理すべきことを考えておく必要がある。最初の予想より多くのデータクラスを扱うことになった場合は、次に示す問いに対する答えがアプリケーションに影響を及ぼす可能性がある。たとえば、ユーザーが閲覧あるいは選

択した情報への参照を得る方法を知っているか？また、ある種の状況においてのみ、ある種のルートスクリプトが現われるべきか？たとえば、先ほどのサンプル・アプリケーションにおいて、“飼料”ルートスクリプトは、ユーザーがラマ・ビューアのアイテムを選択すると現われるが、牧場主ビューアのアイテムを選択すると現われない、というようなことである。

これらの問いに答えながら、アプリケーション設計の残りを眺めてみると、今後の設計上の決定をより的確なものにすることができるだろう。

ルーティングフォーマット

Newton 2.0では、印刷フォーマットとプレルーティング初期設定は、ルーティングフォーマット（routing formats）というオブジェクトの中にカプセル化される。これらのオブジェクトは、アプリケーションの外部で使用できるよう、データをフォーマットする。本節では、さまざまなルーティングフォーマットのタイプとそれらの作成方法を扱う。

印刷とファクスについては、protoPrintFormatに基づいてルーティングフォーマットを作成する。作業の大半は、見栄えのよいビューを設計し、正確なビュー位置調整をほどこし、さらにNewtonScriptのメモリの使用をできるだけ抑えるということになろう。印刷フォーマットコードは、target変数の内容を表示するビューを作成しなければならない。注：targetへ書き込んだり、fields.bodyにアクセスしてはならない。どちらの挙動も定義されないためである。

フォーマットのPrintNextPageScriptメソッドは、ルーティングすべきデータがまだある間は、非nil値を戻さなければならない。フォーマットのPrintNextPageScriptがReDoChildrenビュー・メソッドを呼び出し、そのビュー・メソッドが新たな情報により子ビューを再生するように設計すればよい。あるいは別の方法として、ビューの内容を直接更新するやり方がある。たとえば、SetValue(myParagraph,'text, newText)のようなコードにより、テキストビューの内容を変更すればよい。

ファックス・プロトコルには、数秒間、非アクティブな状態が続くと、タイムアウトするものがある。時間のかかる計算をするとか、プリントレイアウト用に複雑な図形を描画しなければならない場合、フォーマットのFormatInitScriptメソッドを使用する。このメソッドは、ファックス機に接続する前に必ず呼び出される。

'frameまたは'textのdataType（たとえば、BeamまたはMail）をサポートするトランスポートを機能させるには、protoFrameFormatに基づいてルーティングフォーマットを作成する。単純なNewtonScriptフレームをルーティングする場合、アプリケーションが特別なプレルーティングのフォーマットを実行する必要はないかもしれない。しかし、この方法でフォーマットを登録するには、BeamあるいはMailなどのトランスポートがフレームまたはテキスト情報を移動できることを、システムに通知しなければならない。

デフォルトでは、protoFrameFormatはframeとtext両方のdataTypeを扱う。テキストオンリーあるいはフレームオンリーのフォーマットが必要ならば、フォーマットのdataTypesスロットをオーバライドすればよい。たとえば、ルーティングフォーマットが、NewtonScriptフレームの送信のみをサポートしている場合、次のようになるだろう。

```
LlamaFrameFormat := {
  _proto: protoFrameFormat,
  symbol: kFormatSym,
  title: kFormatTitle,

  // override if you don't want both 'frame & 'text
    dataTypes: ['frame]
}
```

フォーマットからトランスポートへ

よくある質問にこういうものがある。“システムは、どうやってActionリストに表示するトランスポートを決定するのか？”簡単に言うと、Actionボタンが、データをルーティングできるルーティングフォーマットを探し、次にそのフォーマットの中の少なくとも一つを処理できるトランスポートを探すのである。ユーザーがActionボタンをタップする前と後には、どのようなことが起きているのか、さらに詳しく説明しよう。

1. デベロッパがNTKにおいてルーティングフォーマットを作成する。ベースビュー内のスロットがこれらのフォーマットを参照することになる。

2. アプリケーションが、これらのフォーマットをRegisterViewDef (format, dataClassSym)を用いてinstallScript内に登録する。次のようなコードでフォーマットにアクセスする。

   ```
   partFrame.theForm.myBaseViewSlot
   ```

   （注：これを行うためにNTK関数のGetLayoutを使ってはならない。詳細についてはDTS Routingサンプルを参照のこと。）

3. ユーザーがアプリケーションをオープンし、データの閲覧と選択を行う。これらのデータをターゲットと呼ぶ。

4. ユーザーがActionボタンをタップする。

5. Actionボタンが、ルーティングすべき**データ**を決定する。Actionボタンは、継承を使用して:GetTargetInfo('routing)を探索する。その結果、{target: ..., TargetView: ...,}などのフレームが戻される。たとえば、ユーザーが先ほどのアプリケーションのラマ牧場主を閲覧しているとすれば、ターゲットは次のようなフォームのフレームになるかもしれない。

   ```
   {class: '|rancher:jX|, rancherID: 929, name: {...}}
   ```

   （注：アプリケーションが既にtargetおよびtargetViewスロットを保持しているならば、GetTargetInfoは省略可能である。）

6. Actionボタンが、ルーティングすべき**データのクラス**を決定する。システムは、ターゲットのデータクラスを決定するために、ClassOf 関数を使用する。

7. Actionボタンが、このデータクラスの表示またはルーティングが可能な**フォーマット**を決定する。Actionボタンは、データのデータクラスとビュー定義レジストリを用いて、そのデータクラスの登録済フォーマットのリストを作成する。ビュー定義レジストリは、先にRegisterViewDefで登録したフォーマットを含んでいる。（注：このリストは、ルーティグフォーマットでないオンスクリーン・ビュー定義を含まない。詳細については、この記事の“Routing Gotchas”の節を参照のこと。）

8. Actionボタンが、これらフォーマットのうち少なくとも1つを表示できる**トランスポート**を決定する。トランスポート、フォーマットの両方ともdataTypesスロットをもち、各スロット内の少なくとも1つのdataTypeが、出現するトランスポートに適合していなければならない。たとえば、このステップにおいて唯一可能なフォーマットが、カスタムのLlamaFrameFormatで、そのdataTypesスロットが['frame]だとすると、dataTypes配列に値'frame をもつトランスポートのみが、Actionリストに含まれることになる。

9. Actionボタンが、追加すべき**ルートスクリプト**を決定する。Actionボタンは、継承を使用して、ルートスクリプトの配列を含むrouteScriptsスロットを探索する。ルートスクリプト・フレームは次のフォームをもつ。

   ```
   {routeScript: 'myExplodeScript, title: "Explode", icon: kMyIcon}
   ```

   （注：ルートスクリプトを動的に決定する必要がある場合は、*Newton Programmer's Guide* の GetRouteScriptsビューメソッドに関する項を参照のこと。）

10. Actionボタンが、新たに作成されたリストを表示する。このリストでは、トランスポートとルートスクリプトが線で区切られている。

11. リストの中からトランスポートを1つ選択すると、Actionボタンがカレント・フォーマットのSetupItemメソッドを呼び出し、次にトランスポートのカスタム・ルーティングスリップをオープンする。ユーザーがフォーマットを切り替えると、システムが新たなフォーマットのSetupItemメソッドを呼び出す。

12. ルートスクリプトを1つ選択すると、Actionボタンはそのメッセージを自分自身へ送る。先ほどのルートスクリプト例を選択したとすると、Actionボタンは次のようなコードを実行する。

```
self:myExplodeScript(target, targetView);
```

複数アイテムターゲット

アプリケーションがオーバービューあるいは複数選択を扱うならば、ルーティングコードは、複数アイテムターゲットに対応していなければならない。複数アイテムターゲットに関わるオブジェクトは2種類あるため、この語は混乱を招きやすい。複数アイテムターゲットとは、複数のオブジェクトに関する情報を含む（そして、スープへ入れることができる）特殊な配列である。これは、スープエントリ・エイリアスとしてストアされるスープエントリを含むこともある。複数アイテムターゲットを作成するには、CreateTargetCursor関数を用いる。

複数アイテムターゲットの内容を読むには、GetTargetCursor関数を呼び出す。この関数は複数アイテムターゲットを得ると、ターゲットカーソルを戻す。ターゲットカーソルとは、基本的なスープカーソル・メッセージであるNext、Prev、およびEntryに応答するオブジェクトであり、アイテムがもはや存在しないと、nilを戻す。もしカレント・アイテムがスープ・エイリアスとしてストアされていれば、ターゲットカーソル・メソッドは、エントリ・エイリアスを変換してスープエントリを戻す。

開発中のアプリケーションがオーバービューをもつならば、ルートスクリプト関数は複数アイテムターゲットを扱わねばならない。TargetIsCursor関数を用いれば、ターゲットが複数アイテムターゲットかどうか確認できる。たとえそのターゲットが複数アイテムターゲットでなくても、GetTargetCursor(item)は、1つのアイテムを含むターゲットカーソルを正しく戻す。ルートスクリプトにおけるGetTargetCursor(item)の使用法の一例を次に示す。

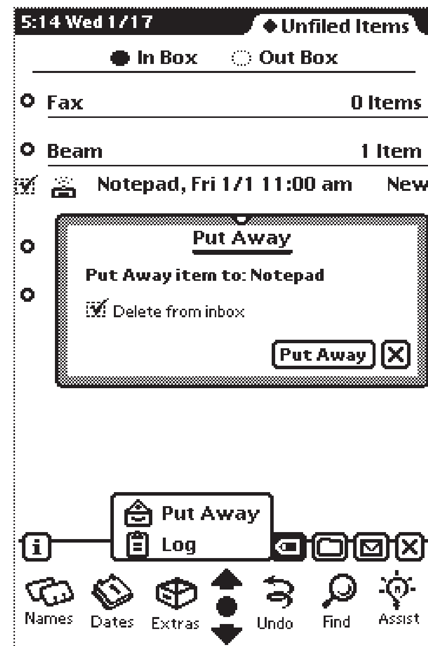
```
myDeleteScript := func(target, targetView)
  begin
    local current;
    local tc := GetTargetCursor(target, nil);
    while (current := tc:entry()) do
      begin
        self:HandleMyDelete(current);
        tc:Next();
      end;
  end;
```

ルーティングフォーマットは、複数アイテムターゲットのサポートを表示する2つの別個のフラッグをもつ。storeCursorsスロットは、アイテムが送信箱（Out Box）内にどのようにストアされているかを決定する。フォーマットのstoreCursorsスロットがnilならば、複数アイテムターゲットは、複数アイテムターゲットとしてストアされずに、別個の送信箱（Out Box）アイテムに分かれている。また、たとえstoreCursorsスロットがtrueでも、複数アイテムターゲットをサポートしないトランスポートは、複数アイテムターゲットを別個の送信箱（Out Box）アイテムに分ける。storeCursorsのデフォルトは、protoPrintFormatに対してはtrue、protoFrameFormatに対してはnilである。

usesCursorsフラグは、印刷フォーマットがどのようなデータを扱うように設計されているかを示す。印刷フォーマットのusesCursorsスロットにnil（デフォルト値）が設定されていれば、印刷フォーマットは各アイテムにつき1度作成される。あるアイテムの印刷が終わると、targetの値は次のアイテムの値に変わり、システムは印刷フォーマットを再度作成する。usesCursorsが非nilであれば、印刷フォーマットは複数アイテムターゲットを扱うので、GetTargetCursorを使用して、それらのアイテムに対し反復適用を行わねばならない。

その他のルーティング・フック

アイテムを片付けているときに、複数アイテムターゲットを受け取ることもあろう。受信箱（In Box）においては、ユーザーはアイテムを選択し、次にトランスポートボタン（荷札アイコン）をタップし、さらにアイテムを受信箱（In Box）からその他のアプリケーションへ片付けることができるのである。



たとえば、友人が赤外線ビームで名刺を送ってくると、そのアイテムをNamesアプリケーションへ片付けることができる。アプリケーションにPut Away（片付ける）をサポートさせるためには、1つの引き数、すなわちアイテムをとるPutAwayScriptメソッドをベースビュー内に用意しなければならない。このメソッドは、アプリケーションがそのデータを扱えることを確認し、適切なアクションを実行した後に、正常終了したことを示すtrueを戻さなければならない。たとえば、Namesアプリケーション用のPutAwayScriptの場合は、アイテムが有効な1つのNamesカードであることを確認し、Namesスープエントリを作成した後に、正常終了すればtrueを戻す（成功しなければ、nilを戻す）。 アプリケーションがクローズしているときにPutAwayScriptを呼び出す可能性もあることに、注意しなければならない。

ルーティングにprotoActionButtonを利用するのであれば、送信箱（Out Box）へ送られるアイテムは、通常appSymbolスロットをもつ。このappSymbolスロットには、ActionボタンによりappSymbolのカレント値が設定される。ベースビュー内のappSymbolスロットを探索するには、継承を使用する（appSymbolスロットをまだ追加していなければ、NTKがそれをベースビュー内に作成する)。ユーザーがappSymbolを含むアイテムに対してPut Awayを選択すると、そのアプリケーションがインストールされ、さらに送受信箱（In/Out Box）が作成したPut Awayスリップ内の選択肢として表示される（先の図を参照のこと）。

ある種のデータクラスの情報を片付けることが可能である、とシステムに対して通知するためには、もう1つ別な方法がある（この方法は、他のアプリケーションがデータの作成またはルーティングを行えるのであれば、必要となる可能性がある）。この方法は、RegAppClasses関数を用いて、データクラス・シンボルをinstallScript内に登録する。2つ以上のアプリケーションが1つのアイテムを片付けられるのであれば、Put Away（片付ける）スリップはPickerを表示して、ユーザーが宛先のアプリケーションを選択できるようにする。

Send関数もまた便利な関数である。protoActionButtonとActionリストを使わずに、送信箱（Out Box）へアイテムを送るようにしたいアプリケーションもある。こういう場合には、どのトランスポートあるいはトランスポートグループが適しているのかをまず決めなければならない。トランスポートグループとは、同種のトランスポートの集まりで、一度にアクティブになれるのは、そのうちの1つのみである。これらト

ランスポートグループはシンボルにより識別される。その一例として、'mailグループがある。たとえば、誰かに自動的にe-mailを送るボタンが欲しいとする。そのボタンは一例として次のようなコードを使用する。

```
Send('mail, {body: yourData, toRef: [aRecipient]}).
```

アイテムへ追加すべき重要なスロットとしては、データ（target）含むbodyスロットと、toRefスロット内の受信者がある。また、titleスロット（主題行）とappSymbolスロット（Put Away用）を追加することもできる。アイテムへ追加できるその他のスロットのリストについては、Newton Programmer's Guide (NPG)を参照していただきたい。ただし、ある種のアイテムのスロットについては、すべてのトランスポートに適用できるとは限らない。

toRefスロット内の受信者は、nameRefsの配列により表現する。nameRefsは、protoListPicker（これは、ルーティングスリップ内の共通の“to”行と“cc”行を含む）に基づいてチューザから戻される受信者フレームである。Namesスープエントリへの参照があり、それをnameRefへ変換したい場合は、nameRefのデータ定義を用いなければならない。nameRefのデータ定義とは、特定クラスのnameRefの処理方法を記述するオブジェクトである（たとえば、|nameRef.fax|と|nameRef.email|）。どのトランスポートも、その優先するnameRefクラスシンボル（そのアドレッシングクラス）を、adressingClassスロット内にストアしている。

たとえば、手紙をファックスするプログラムをコードしたいとする。Namesファイル内にない人へ送りたければ、アドレッシングのための最小スロットを含むフレームで代用できる（アドレッシングクラスとトランスポートにより異なる）。手紙をファクスするには、次の例のようなコードを使えばよい。

```
faxLetter := func()
begin
  local transportSym := '|faxSend:Newton|;

  rancher := {
    name: {last: "Bell", first: "J. Christopher"},
    phone: SetClass("408 555 1212", '|string.fax|)
    };

  target := {class: '|letter:jX|, style: 'VisitUsAgainSoon};

  // TransportNotify sends a message to our transport
  // to create a new item that we pass to Send(...)
  item := TransportNotify(transportSym, 'NewItem, [nil]);

  aClass := GetItemTransport(item).addressingClass;

  item.body := target;
  item.toRef := [GetDataDefs(aClass):MakeNameRef(rancher,
          aClass)];

  // Register a format with RegisterViewDef (see rest of article...)
  // It must have this symbol in its symbol slot.
  item.currentFormat := kMyViewFormatSym;

  Send(transportSym, item); // submit the item to the outbox
end;
```

テキストをサポートするトランスポートでSendを使用する場合は、そのトランスポートは、ルーティングフォーマットのtextScriptメソッドを呼び出すことにより、アイテムをテキストへエクスポートしなければならない。textScriptメソッドは、item.body（itemは、textScriptメソッドへ渡す引き数の一つである）を変換し、送るべきテキストを戻す。

どこにコードを書くべきか？

ルーティング・フックのいくつかは、メソッドあるいは変数を探索するのに継承を使用する。もっと具体的に言えば、Actionボタンがそのメッセージ・コンテキスト（その_protoおよび_parentチェーン）（注：式やメソッドを評価するときのフレーム＝コンテキストフレーム）を使用して、メッセージを自分自身へ送る。たとえば、ベースビュー内にGetTargetInfoメソッドを書く場合、Actionボタンはペアレント継承によりベースビュー・メソッドを探索する。しかし、GetTargetInfoが実行されるときには、selfがActionボタンになる。これらのメッセージをベースビュー内に入れるデベロッパは多いが、ルーティングのコンテキスト依存性は、コードを書き込む場所に関するフレキシビリティをもたらす。たとえば、サブビューが異なる情報をルーティングする、あるいは情報を違った方法で扱うようにしたい場合は、変数とメソッドをサブビュー内に書くことができる。

フォーマットとアプリケーションは別々の場所に存在することに注意しよう。Actionボタンがルーティングフォーマットを探索するのは、ルーティングフォーマットがグローバルに登録されているためである（先に述べたRegisterViewDefを参照のこと）。このアーキテクチャを利用すれば、自分のアプリケーションを他のデベロッパに拡張させることができる。自分のデータのフォーマットを公開すれば、そのデータクラス用の新たなフォーマットを他のデベロッパがグローバル登録することができる。たとえば、protoPrintFormatを'personクラス上に登録すれば、Namesアプリケーション（および'personクラスを使用するその他のアプリケーション）のユーザーが、その新たなフォーマットを、PrintあるいはFaxルーティングスリップ内の選択肢として見ることができる。

ルーティングに関する一般的な質問	関連コードを書くべき場所
ターゲットは何か？	Actionボタン・メッセージ・コンテキスト
可能なルートスクリプトはどれか？	Actionボタン・メッセージ・コンテキスト
routeScriptsはどこに書くか？	Actionボタン・メッセージ・コンテキスト
可能なルーティングフォーマットはどれか？	ViewDefレジストリ
どのトランスポートタイプを使用すべきか？	ViewDefレジストリとフォーマットのdataTypesスロット
データを視覚的に表現する方法は？	ルーティングフォーマット（protoPrintFormatに基づく）
テキストを送る前にどこへエクスポートするか？	ルーティングフォーマット（TextScriptメソッド）
データを送る前にどこで処理するか？	ルーティングフォーマット（SetupItemメソッド）
時間のかかるプレファクスコードはどこで実行するか？	ルーティングフォーマット（FormatInitScriptメソッド）
PutAwayScriptはどこに書くか？	アプリケーションのベースビュー

組込みトランスポートへのルーティングのためのチェックリスト

ルーティング要件

- routeScriptsスロットを追加 (†)
- GetTargetInfoメソッドを追加（†）
- Actionボタンを追加（protoActionButton）
- [Un]RegisterViewDef呼出しをinstallScriptとremoveScriptへ追加
- RegisterViewDefを用いて、partFrame.theForm.myBaseViewSlotのコードでフォーマットへアクセス（注：これを行うために、NTK関数のGetLayoutを使ってはならない。詳細については、DTS Routingサンプルを参照のこと。）

Print、Fax、および将来のビュー・トランスポートの要件

- NTK内のprotoPrintFormatに基づき、レイアウトを作成
- 一意的なフォーマット・シグネチャを含むsymbolスロットを作成
- フォーマット・タイトルを含むtitleスロットを作成
- 子ビューを追加（サブビューを引き出す、あるいはviewSetupChildrenScriptを使用）
- printNextPageScriptメソッドを追加
- プレルーティング・セットアップが必要ならば、SetupItemメソッドを追加
- プレファクス初期設定に時間がかかるならば、FormatInitScriptメソッドを追加
- NTKのGetLayout関数を用いて、ベースビュー・スロット内にフォーマットへの参照をおく。これをinstallScriptから参照することになる（先に述べたRegisterViewDefを参照のこと）。

Beam、Mail、およびその他のフレームまたはテキストのトランスポート

- protoFrameFormatに基づき、フォーマットを作成
- 一意的なフォーマット・シグネチャを含むsymbolスロットを作成

- フォーマット・タイトルを含むtitleスロットを作成
- テキスト・エクスポートをサポートするため、TextScriptメソッドを用意
- プレルーティング・セットアップが必要ならば、SetupItemメソッドを追加
- ベースビュー・スロット内にフォーマットへの参照をおく。これをinstallScriptから参照することになる（先に述べたRegisterViewDefを参照のこと）。
- Put Awayをサポートするためには、[Un]RegAppClasses呼出しをinstallScriptとremoveScriptへ追加
- Put Awayをサポートするためには、PutAwayScripメソッドをベースビューへ追加

(†) これらの追加はしばしばベースビュー内に行なわれる。ただし、Actionボタンからのコンテキストに依存する。

ルーティングの落とし穴

アプリケーションにルーティングを書く際に心得ておきたい、よくあるミスあるいは落とし穴（“gotchas”）をいくつか取り上げる。

ルーティングフォーマットとオンスクリーン・ビュー定義（viewDefs）の違いがわからない人は多い。これらは、登録あるいは登録の取消しに同じ関数を用いるが、それぞれ用途が異なる。オンスクリーン・ステーショナリとviewDefsについては、NPGのStationeryの章を参照していただきたい。オンスクリーンviewDefsはルーティングフォーマットとして使えないし、またルーティングフォーマットもオンスクリーンviewDefsとしては使えない。これは、viewDefsのタイプを示す typeスロットのせいである。ルーティングフォーマットについては、protoRouteFormatに基づいているため、そのタイプはprintFomatまたはrouteFormatになる。ルーティングフォーマットと、viewerあるいはeditorなどのオンスクリーンviewDefsタイプを区別するために、このようになっている。

レイアウトをオンスクリーン・ビューと印刷の両方に使いたいのであれば、2つの別個のレイアウトを作成しなければならない。しかし、この2つのレイアウトは似ているため、ほどんど同じ2つのビューをレイアウトするだけですむ。開発時間を節約するために、共通の挙動をカプセル化するユーザープロトを作成し（NTKユーザーマニュアルを参照のこと）、これをオンスクリーン・ビューと印刷フォーマットのベースとして使ってもよい。印刷フォーマットは、表示データを含むtargetスロットを自動的にセットアップするということが、唯一のgotchaである。ユーザープロトの挙動を標準化するために、オンスクリーン・ビューに対し変更を加えて、オンスクリーン・データを含むtargetスロットを作成する必要があるかもしれない。

もう一つのよくあるのが、データクラスを誤って設定するという問題である。GetTargetViewメソッドがtargetInfoフレームを戻すとき、targetInfoフレームは、意味のあるクラスのターゲットをもっていなければならない。つまり、classof(target)は、データクラスを表現する一意的なシンボルを表現する。ターゲットは通常フレームなので、ターゲットはclassスロットをもたねばならないということである。あるいは、GetTargetInfoメソッドによりtargetが戻されたときにクラススロットが追加されるということである。この記事の“フォーマットからトランスポートへ”の節で述べた通り、Actionボタンは、可能なルーティングフォーマットとトランスポートを決定するために、データのクラスを使用する。

もう一つの落とし穴は、オーバービューからルーティングを機能させることである。ある種のアプリケーションについてオーバービューと複数選択をより簡単にするような、ルーティングにおける特殊コードが存在する。開発中のアプリケーションにおいて、フォーマットを'frame、'text、'viewのタイプにのみ登録するようにし、しかも印刷フォーマットが特殊な初期設定を必要としなければ、特殊な'newtOverviewデータクラスを使って、オーバービュー内に複数アイテム選択を表現できる。

GetTargetInfoメソッドが複数アイテムターゲットを戻すのであれば、CreateTargetCursor('newtOverview, myItems) コードを用いて、特殊データクラスをもつ複数アイテムターゲットを作成できる。こうすると、開発中のアプリケーションが、'newtOverviewクラス上に登録された（このデータクラス上に独自のviewDefsを登録してはならない）組込みルーティングフォーマットを使用できるようになる。このクラス上に登録されたフォーマットのデフォルトは、nilに設定されたusesCursorsスロットをもつので、システムは、この記事の“複数アイテムターゲット”の節で述べたような、便利なデフォルトのルーティングフォーマット挙動を利用できる。これは、複数アイテムの印刷とルーティングのために多いに役立つはずである。

このやり方にはいくつかの制約がある。先に述べたように、ファクス接続中のタイムアウトを回避するため、印刷フォーマットが、SetupItemメソッドあるいはformatInitScriptメソッド内で特殊な初期化を必要とする場合もあろう。もしそうならば、複数アイテムターゲットに対し'newtOverviewクラスを用いたとき、印刷フォーマットは、SetupItemまたはFormatInitScriptメッセージを得るとは限らないことに注意していただきたい。

'frame、'text、'view以外のdataTypeを使用したい場合、あるいは印刷フォーマットがSetupItemまたはFormatInitScriptメッセージを必要とする場合には、CreateTargetCursorを用いて複数アイテムターゲットを作成するときに、独自のデータクラス・シンボルを使わなければならない。たとえば、先に例をあげたアプリケーションにおいて、ラマのレイアウトから印刷とファクスは可能だが、赤外線ビームやメールなどのその他のトランスポートは不可能だとしよう。デフォルトの'newtOverview挙動が赤外線ビームやメールなどのトランスポートを可能にする。それは、フォーマットを'frameと'text dataTypesで登録するためである。したがって、'newtOverview挙動は利用できない。その代わり、次のようなコードを用いれば、複数アイテムターゲットを作成することができる。

```
CreateTargetCursor(kLlamaClassSym, selectedItems);
```

先の例のkLlamaClassSymデータクラスについては、protoPrintFormatに基づくルーティングフォーマットのみを登録すれば、Actionリスト内のトランスポート選択肢として、PrintとFaxのみが表示される。

ルーティングフォーマットが複数アイテムを扱うようにしたい場合は、そのusesCursorsスロットをtrueに設定すれば、1ページ上に複数アイテムを印刷したり（GetTargetCursorを用いてアイテムリストを縦覧する）、複数アイテムを別の方法でルーティングすることができる。たとえば、protoFrameFormatに基づくフォーマットならば、SetupItemメソッド内のアイテムのリストを縦覧し、すべてのアイテムを表わす新たな仮想バイナリ・オブジェクト（VBO）をitem.bodyに設定できる（詳細については、NPGのData Storageの章を参照のこと）。システムは、VBOをNewtonScriptメモリではなく、ストア上に格納するため、NewtonScriptメモリ不足を発生させることなく、1つのOut Boxアイテム内に大量のデータをルーティングできる。仮想バイナリ・オブジェクトには、意味のあるクラスシンボルを必ず与えること。そうすれば、アプリケーションは、Put Away中にこのデータクラスをチェックできる。

アプリケーション設計に、オプションのNewtAppアプリケーション・フレームワークを使っているデベロッパもいるだろう（詳細については、NPGを参照のこと）。アプリケーションがNewtAppフレームワークを使用しており、デベロッパがnewtOverviewと呼ばれるレイアウトプロトを使っていて、先に述べた'newtOverviewの複数アイテム挙動を利用したくない場合は、余分な作業が必要となる。すなわち、複数アイテムターゲットのデータクラスとして使われるシンボルを含むレイアウトに（あるいは、その_parentチェーンのどこかに）、overviewTargetClassスロットを用意しなければならない。

以上、Newton 2.0アプリケーションのルーティングを書く前に心得ておくべき基本事項を解説した。さっそくルーティングにとりかかっていただきたい。これらの概念をマスターすれば、どんな情報でも移動させることができるはずである。

（著者あとがき：J. ChristopherはMaurice Sharp とJohn Perry に対し、本記事執筆上の支援を感謝する。）

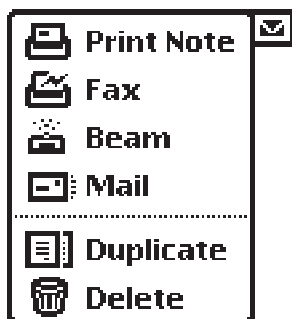
NTJ

トランスポートを書く

by J. Christopher Bell, Apple Computer, Inc.

トランスポートとは何か？

おなじみの封筒アイコン、つまりActionボタンをタップすると、情報を移動する方法のリストがオープンする。



The items above the line in the Action list are serviActionリスト内の線より上のアイテムは、情報をNewtonデバイスの外部へ移動するサービスである。たとえば、BeamでExtras drawerからパッケージを送信したり、Printでノートを印刷したり、Mailで友人にアイテムを送信したりできる。これらの通信サービスをトランスポートと呼ぶ。組込みトランスポートも用意されているが、ユーザーがパッケージとしてインストールできるような、新たなトランスポートを作成してActionリストに選択肢を追加してもよい。ある意味で、トランスポートは、複雑な通信コードに対するユーザーインタフェースと言えるかもしれない。この記事では、Newton 2.0のトランスポートを書く前に心得ておくべき点をいくつか取り上げる。

この記事を読む前に、ルーティングという語を理解しておく必要がある。ルーティングとは、 Actionボタンと送受信箱 (In/Out Box) を使って情報を移動すること全般をさす用語である。ルーティングの詳細については、Newton Programmer's Guide (NPG)と、Newton Technology Journal今月号の“ 情報を移動してみよう ”の記事を参照していただきたい。

理解しておかねばならない最も重要な語はdataTypeである。dataTypeとは、情報をNewtonデバイスの外部へ送信する方法の総称的分類である。一般的なdataTypeは、普通テキスト（'text dataType)、NewtonScriptフレーム（'frame dataType)、およびイメージング・レイアウト（'view dataType）である。独自のdataTypeを定義することもできるが、この機能を利用するためには、アプリケーションにそのカスタムdataTypeの詳細を知らせなければならない。この問題に関する詳細については、“ 情報を移動してみよう ”の記事を参照していただきたい。

ルーティングフォーマットの基本についても理解しておかねばならない。印刷フォーマット・レイアウトとプレルーティング初期設定は、ルーティングフォーマットというオブジェクトの中にカプセル化される。これらのオブジェクトは、アプリケーションの外部で使用できるようにデータをフォーマッティングする。

トランスポートを書くときは、protoBasicEndpointによる、エンドポイント通信コードの設計、コーディング、およびテストに時間の大半を費やすことになろう。エンドポイントコードの設計およびデバッグに関するTipsについては、NPG 2.0とDTS Communicationsサンプルを参照していただきたい。

基本通信コードを設計したら、それを次の手順でテストする。まずprotoTransportを作成し、RegTransport関数を用いてそれを登録し、エンドポイントコードをSendRequestとReceiveRequestメッセージにフックする。これらについては後で解説する。トランスポートがデータを送信できたら（あるいはデータ受信のみかもしれない)、ルーティングスリップを設計し、受信者とオプションを変えてコードをテストしてみる。この記事では、トランスポートを書くための入門として、設計上のポイントと基本的なトランスポートAPIを取り扱う。

トランスポートを書く前に

開発中の通信サービスがBeamまたはMailに似ているのであれば、Newtonトランスポートのようにコードを組むことになろう。しかし、すべての通信コードがトランスポートを作成するのに適しているわけではない。トランスポートが開発中のプロジェクトに最適かどうか、時間をかけて決定したい人もいるだろう。すべての状況に適用できるとは限らないが、トランスポート設計のためのガイドラインを次に示す。このガイドラインを読む際に注意して欲しいのは、トランスポートはデータ送受信の両方を行う必要はないことである。受信のみあるいは送信のみでもかまわない。

トランスポート・ガイドライン

次の問いに対する答がすべて“ yes ”ならば、トランスポートは適切なアプローチと言えよう。

- エンドポイントコードが、組込みトランスポート（たとえば、ルーティングスリップ、送受信箱 (In/Out Box)の使用、ステータス・ビューなど）に似たユーザーインタフェースをもつのか？
- 送信するアイテムのキューイングが意味をなすのか？
- すべての通信セットアップ・プレファレンスが送受信箱 (In/Out Box)に属すのか？
- 送受信箱 (In/Out Box)からSendまたはReceiveを呼び出すことが意味をなすのか？

次の項目にあてはまれば、トランスポートとしてコードを組まないほうがよいかもしれない。

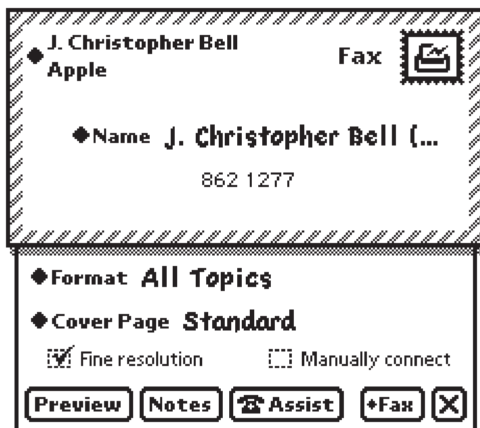
- 上記のトランスポート・ガイドラインの中で、“ No ”と答えた項目が1つでもある。
- 入出力情報が、プレビューしたりキューに入れたりする別個のアイテムではなく、ストリームである。
- コードの目的がNewtonスープと、非Newtonデータベースまたはアプリケーションを同期させることである。その場合は、1つのアプリケーション内で同期化アクションを実施するほうが､簡単である。
- そのトランスポートを、自社製アプリケーションにのみ使用可能なものにしたい。（下記の注意を参照のこと）
- 他のデバイスへの接続時に、トランスポートが送受信箱（In/Out Box）のアイテムを動的に処理しなければならない。リモートアイテムを使用するトランスポートが設計可能である。つまり、In Box

（Out Boxではなく）のアイテムが、ユーザーによる閲覧やオープンが可能な付加アイテムを提供するサービスから、概要の情報をダウンロードできる。しかし、エンドポイントコードが、アイテムをOut Boxへ追加するか、あるいは送受信箱（In/Out Box）のレコードを動的に追加または削除する必要があるならば、アプリケーションとしてコードを設計した方が、よりユーザーの期待に沿うものになろう。

- トランスポートが、たとえばページ上にビューを描画するというように、PrintまたはFaxに似ている。その場合は、プリンタドライバを書く必要があろう。

注意：トランスポートを自社製品のみに使用可能なものにしたいデベロッパもいるかもしれない。そうするのが一番良い場合もあるが、常にそうとは限らない。セットアップとアクティビティ（プレファレンス、Send/Receiveの初期化）は、それらのサービスを制御するアプリケーション内で行うことが、最もストレートなユーザーインタフェースかも知れない。1種類のアプリケーションでのみ機能するトランスポートを書く代わりに、エンドポイントコードを1種類のアプリケーション内にカプセル化するか、あるいはUnitインポート/エクスポート機能を利用して、数種のアプリケーションでコードを共有するようにしてもよい。Unitインポート/エクスポート機能については、DTS Q&Aを参照のこと。

先に述べたガイドラインによると、開発中のトランスポートは他のトランスポートと異なるのではないか、と感じる場合もあろう。もしそうなら、そのトランスポートには、やや異なるユーザーインタフェースが必要かも知れない。たとえば、トランスポートが小さなOut Boxアイテムを大量に作成するのであれば、NPGでhiddenスロットに関する情報を確認していただきたい。このスロットを用いると、Out Box内のアイテムを見えない状態にできる。



トランスポートを自社製アプリケーションでのみ使用可能にするには、固有のルーティングフォーマットのみがサポートできる、カスタムdataTypeを（ルーティングフォーマットのdataTypeスロット内に）作成する。たとえば、'frameあるいは'textなどの標準的なdataTypeを使用せずに、`'|MyDataBaseRecord:MYSIG|`などとすればよい。

ルーティング・スリップ

基本的なルーティングスリップは、便利な組込みprotoFullRouteSlipを利用すると、たやすく設計できる。これには、ルーティングスリップ内のほとんどのユーザーインタフェース・エレメントが含まれている。たとえば、Format Picker、封筒アイコン、戻りアドレスプロト、クローズボックス、Sendボタンなどである。ルーティングスリップの最も複雑な設計決定には、受信者情報がかかわってくる。ルーティングスリップの上部分は受信者制御を含み、下部分はフォーマッティング制御を含むことに注意すること。

開発中のトランスポートがBeamに似ているならば、プレファレンスがSend Laterに設定されているときは受信者制御が不要なので、簡単である。大部分のトランスポートについては、誰がアイテムを受信すべきなのかを、ユーザーがトランスポートへ伝えるように設計しなければならない。

受信者を選択するための一般的なユーザーインタフェースは、protoAddressPickerである。これにより、ユーザーは、メッセージの“to”や“cc”の所に入れる受信者をNamesファイルから選択できる。トランスポートがMailの使う受信人タイプを使用するならば、特別なことをする必要は何もない。電話またはファクス番号に受信人タイプを使いたければ、protoAddressPickerのclassスロット上書きすればよい。たとえば、電話番号を選ぶには、`'|nameRef.phone|`を指定する。

組込みクラスがサポートしていないカスタム宛先人情報（たとえば、インターネットアドレスを使えないe-mailなど）が必要な場合は、`nameRef`データ定義の独自のサブクラスを追加する必要がある。`nameRef`の独自のサブクラスを追加する方法については、NPG（protoListPicker、protoNameRefDataDef）とDTSサンプルを参照していただきたい。

ルーティングスリップの下半分には、サブジェクト・エディタや解像度設定などのフォーマッティング制御、あるいはトランスポート独自の他の制御を追加できる。セットアップ制御の大半をトランスポート・プレファレンスに入れることにより、ルーティングスリップの下半分はなるべく単純化することをお勧めする。

ルーティングスリップの下半分のサイズは、場合によって多少変化するため、それがビューレイアウトを難しくしている。たとえば、ただ1つのフォーマット・オプションしかない場合、Format Pickerは見えないため、この部分の高さが変わる。ルーティングスリップの下半分にビューをレイアウトする最も簡単な方法は、ルーティングスリップ・メソッドのBottomOfSlipを用いて、ルーティングスリップの底辺のオフセットを探すことである。

ユーザーがアイテムを送信しようとすると、ルーティングスリップはPrepareToSend messageを受け取る。PrepareToSendメソッドにおいては、ルーティングスリップ制御から情報を取り出し、（Out Boxスープエントリになる）fields内のスロットにその情報をストアし、継承されたメソッドを呼び出してアイテム送信を続行する。受信者情報は、toRef、cc、bccなどの標準スロットにストアする。これらのスロットはnameRefsの配列を含んでいなければならない。`nameRefs`とは、protoListPickerおよびprotoAddressPickerに基づくchooserから戻される、受信者フレームである。

トランスポートグループに関わる、特殊なルーティングスリップ・インタフェースがある。トランスポートを'mailグループのメンバにするには、そのシンボルをトランスポートのgroupスロットに入れればよい。こうすれば、Actionリスト内のトランスポートを、トランスポートのタイトル（たとえば、“MySuperMail”）ではなく、Mailと表示できる。

同一グループ内に1つを超えるトランスポートがインストールされているときは、トランスポートグループ・メンバを切り替えるためにルーティングスリップを用いる必要がある。ルーティングスリップ内のトランスポート・タイトルが、（複数選択肢を示す）ダイヤ印を含むようにすれば、ユーザーはトランスポート・タイトルをタップして、選択したい他のトランスポートグループ・メンバのリストをオープンできる。ユーザーがGroup Transport Pickerを用いてトランスポートを切り替えると、ルーティングスリップがクローズして、新たなトランスポートのルーティングスリップがオープンする。

SendRequestとReceiveRequest

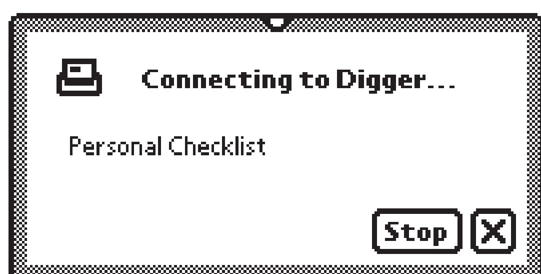
通信コードの大半は、SendRequestまたはReceiveRequestメッセージに応じて実行される。SendRequestにおいては、対応するアイテムを取り出すために、トランスポートへItemRequestメッセージを送る動作を、アイテムに対して必要な数だけ反復適用しなければならない。エンドポイントコードを非同期的に実行し、値がnilになるまで、アイテムに対しItemRequestを呼べばよい。アイテムが完成すると、トランスポートは、transport:ItemComleted(item, state, errorNumOrNil)を用いることにより、Out Boxに対し新たなステータスを通知できる。ステートが'sentならアイテムは Out Box から削除される。エラーが発生したら、ステートとしてnilを渡し、アイテムがOut Box内にとどまる必要があることを指示する。

アイテムを受け取るトランスポートの大半は、別のデバイスへ接続し、できるだけ多くのアイテムを取り出すことにより、ReceiveRequestメッセージへ応答する。ItemComleted関数のステータスを'receivedにして、アイテムをIn Boxへ提出する。リモートアイテムを使用するブラウズ機能を書くこともできる。そのためには、タイトルと受信者情報をダウンロードし、アイテムのremoteスロットに非nil値を設定する。ユーザーは概要の情報のソートや閲覧は可能だが、アイテムをタップすると、引き数causeが'remoteに設定されて、2回めのReceiveRequestが出される。つまり、トランスポートが再度、アイテム全体をダウンロードし、Item Comletedを呼び出さねばならなくなる。リモートアイテムを作成するときは、ユーザーがタップしたときにそのアイテム全体を取り出す方法が表示されるよう、アイテム内のメッセージIDをコード化するとよい。

他のトランスポート・フック

開発中のトランスポートのdataTypesスロットが、'textシンボルを含む（言い換えれば、テキストをサポートしている）ならば、ItemToText関数を使って、アイテムからテキストを取り出せる。ItemToText関数の主要機能は、ルーティングフォーマットtextScriptメソッドを送り、{class: 'text, text: "whatever..."}のフォームで、テキストをアイテムのbodyスロット内にストアする。ItemToText関数の使い方については、DTS Transportサンプルを参照していただきたい。現在、この関数は、NTKプロジェクトに含まれるストリームファイルとして、利用可能である。

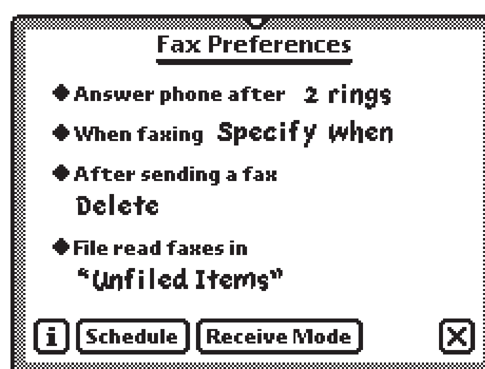
基本的なprotoTransportには、トランスポート・ステータスをユーザーに表示するビューを扱う際に有用なデフォルトが用意されている。



デフォルト・ステータス・ビューの動作の大半は、transport:SetStatusDialog('Connecting,'vStatus,"Connecting to"&&printername&"...") などのコマンドを使って、プログラムできる。開発およびデバッグにおいて、これらのデフォルトを使用したいというデベロッパは多いはずである。そうすれば、エンドポイントコードの作成にエネルギーを集中させられるだろう。SetStatusDialogでダイアログをクローズするには、最初の引き数（status)に'idleを設定するだけでよい。組込みの目盛インディケーター、ページ・インディケーター、あるいは転送インディケーター（Barbar pole）の他に、ステータス・インディケーターが必要な場合は、protoStatusTemplateを用いて、ダイアログを完全にカスタマイズすることができる（詳細については、DTSサンプルおよびNPGを参照のこと)。

送受信箱 (In/Out Box)は、トランスポートに対する主要なユーザーインタフェースである。そこで、トランスポートの開発に利用できるユーザーインタフェース・フックをいくつか紹介しよう。（注：メッセージは、送受信箱 (In/Out Box) アプリケーションそのものへ直接送ってはならない。インプレメンテーションが変化する可能性があるためである。）

送受信箱（In/Out Box）で最も頻繁に使うトランスポート・フックは、送受信箱（In/Out Box）内のInfoボタンによりアクセス可能な、プレファレンススリップである（次の図を参照のこと)。



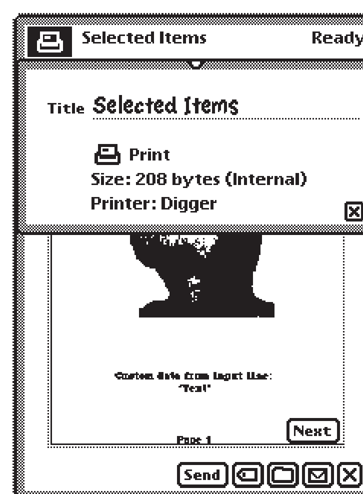
接続オプション、デフォルト値、ならびにログ出力とファイリング・オプションを含むprotoTransportPrefsに基づいて、レイアウトを作成する。プロトには、オプションの制御がいくつか用意されている。

ユーザーが送受信箱 (In/Out Box)のアイテムをタップすると、送受信箱 (In/Out Box)がアイテムビューアをオープンする（次の図を参照のこと。アイテムビューアの中には、いくつかのボタンがある。別のトランスポートへアイテムの再ルーティングを行うActionボタン、場合に応じて表示されるShowボタン、Transportボタン（荷札アイコンがついているボタン）などである。）Transportボタンは、デフォルトのコマンド（状況に応じて、Readdress、Log、PutAwayなど）でPickerをオープンするが、ReplyやForwardなどをトランスポートに追加することもできる。

たとえば、Explodeというトランスポート・アクションを追加するには、トランスポートスクリプト・フレーム（ルートスクリプト・フレームに似ている）の配列を戻すGetTransportScriptsトランスポート・メソッドを書く。たとえば、次のようなフレームにすればよい。

```
{title: "Explode", routeScript: 'myTransportMethod, appSymbol:
kMyTransportSym}.
```

ユーザーが、アイテムビューアの左上隅にあるトランスポートアイコンをタップすると開くような、アイテム情報スリップを設計してもよい。



デフォルト情報にはアイテムのサイズおよび標題が含まれているが、スリップを完全にカスタマイズして、トランスポート固有情報、あるいは In BoxやOut Boxアイテムに関するアイテム履歴を追加することもできる（protoTransportHeaderについては、NPGを参照のこと)。

以上、通信コードを書き、独自のトランスポートを設計する前に心得ておくべきAPIを紹介した。ご健闘を！

NTJ

1ページからの続き

C++ と NewtonScript

ードウェアであるレジスタの操作など非常に低レベルのシステムコードを書くことができる。C++でアルゴリズムを書けば、プロセッサハードウェアの利点を十分に生かすことができる。もっとも大事な点は、Cで書いたコードは、僅かな変更をかけるだけでC++コンパイラによりコンパイルできることである。

パフォーマンスはC++の設計目標で最高の優先順位をあたえられたものの一つである。一般原則として、新機能を加えるに当たり、それを直接使用しないコード化のためのコストがゼロである、つまりパフォーマンスに悪影響がでない場合にのみ、実行された。

Cの後継言語になるためには、C++は既存の開発システムと“高い親和性”を持たねばならなかった。特に、C++コンパイラの出力はCコードとリンクしなけらばならず、C++とCのデータ構造は相互運用可能でなければならなかった。

低レベルのシステムコードとして便利な同一動作は、また偶然誤用される可能性も高い。C++では、プログラミングの誤りによって、実行時間環境が崩壊し、システムクラッシュやデータの破壊をまねくことがよくある。プログラムは手動でメモリの割り当て解除ができるよう設計される必要がある。これは書く際に間違いを犯しやすく、バグがなかなか見つからないことが多いプロセスである。

初期のC++では、例外処理、多重継承、パラメタ化されたクラス、実行時のタイプ情報などはサポートされていなかった。時間をかけて、これらの機能は、パフォーマンスを犠牲にしない、という首尾一貫したやり方で、言語に付け加えられていったのである。C++は今日複雑な言語となり、すべての機能を完全に理解して使いこなすには、数年の経験が必要とされる。

NewtonScript

NewtonScriptの目的は、Newtonアプリケーションのコーディングをできる限り容易にすることである。NewtonScriptのもっとも重要な設計目的のいくつかを列挙すると、以下の通りである。

- Newtonアプリケーションのフレームワークとの自然なやりとり
- 小さいプログラムサイズによる小さいRAMの使用領域
- 実行時の確実・安定性
- 単純性

NewtonScript言語は、Newtonシステムのいたる所で使われているオブジェクトモデルを基盤として構築されている。インラインコンストラクターと組み込みアクセスオペレーターが、オブジェクトの作成と操作を容易にしてくれる。

NewtonScriptは、プロトタイプベースの継承を使用する。これは、ユーザー・インターフェースコードを書くもっとも自然な方法である。ビューシステムは、NewtonScriptのオブジェクトモデルを中心に設計され、システムまたはプログラマが提供するユーザー・インターフェース要素を再利用しやすくしている。

Newton OSは小型デバイスをサポートする目的でつくられているから、NewtonScriptも小容量のメモリ用に設計された。NewtonScript実行可能コードはとても小さく、同等のネィティブコードの何分の一しかない。プロトタイプベースの継承を使用するから、オブジェクトの変化するスロットのみをRAMに入れることにより、RAMの使用スペースを最小に抑えやすい。

NewtonScriptは確実かつ安定した言語である。すべての操作は妥当性をチェックされ、規則違反は制御された例外の原因となっても、システムはクラッシュしない。だから、デバッグははるかに容易となり、アプリケーション間にシステムの“防火壁 (Fire Wall) ”を設ける必要が減る。メモリの割り当て解除は、NewtonScriptが実行時に自動的に行う。このことにより、ぶらさがりポインタや多重割り当て解除など、メモリ管理上のバグがでないし、プログラム設計が単純化される。

NewtonScriptは小型の言語であり、構文や意味も比較的単純である。たいていのプログラマは、ほとんど直ぐコードを読めるし、重要な機能全部を有効利用できるようになるのに、2、3カ月しかかからない。そのような単純性にもかかわらず、NewtonScriptは、完全な汎用言語である。

NewtonScriptの実行モデルは、ハードウェアから独立している。そのかわり、十分に定義された一連のオペレーションをタイプ化されたデータに対して行う。見えない表現へのアクセスは存在せず、それゆえにタイプ間に“危険なキャスト”は存在しない。したがって、NewtonScript言語は安全かつポータブルである、と同時にこのことは、プログラマが最大のパフォーマンスを得るためにハードウェアを充分活用できない、ということを意味する。

C++とNewtonScriptの関係

C++とNewtonScriptは相互に補完する強みを持っている。C++の“Cらしさ”は、ハードウェアの操作や（例えば、デバイスドライバ）高速なコードを書くのに適しているが、安全でない機能がでてきたり柔軟性がないなど、マイナス面もある。NewtonScriptは、その安全性、表現力、ならびに柔軟性ゆえに、高レベルのアプリケーションコードを書くのに適している。反面、NewtonScriptは、パフォーマンスがその分犠牲になり、既存のCおよびC++ソースコードとの互換性がない。

Newton Toolkitに含まれる、ネィティブのNewtonScriptコンパイラは、多くのプログラムでC++のコードとほぼ同レベルのパフォーマンスを与える。NewtonScriptオブジェクトに基礎をおくアルゴリズムは、通常CやC++などとも速度にあまり違いはない。ネィティブのC++のデータ構造を使用するC++コードなら、普通はC++の方が早いだろう。

内部的にみれば、Newton OSは常にC++とNewtonScriptの両方の言語を利用してきた。NewtonScriptは、高レベルで、ユーザー・インターフェースオブジェクトや組み込み型アプリケーションを作成し、システム統合を達成するのに使われる。C++は、低レベルで、OSのカーネル、グラフィックスの図形要素、デバイスドライバ、ならびにNewtonScriptインタープリタの作成に使われる。どちらの言語を使うかは、システム構成要素の要件により決まる。パフォーマンスが致命的に重要ならC++を用いるし、安全性やメモリの省スペース化がもっと大事ならNewtonScriptを用いる。

上記のことを読者はすでにご存知かもしれない。高レベルのオブジェクトモデルを使用してシステム構築をするのは、今や主流となってきているからである。SOMやOLEは、低レベルのコードを統合するシステムワイドなオブジェクトモデルの一例である。SOMやOLEオブジェクトは、C++やその他の言語で書くことができる。オブジェクトの使用者はどの言語が用いられているかを知る必要はない。

Newton OSは、NewtonScriptのオブジェクトモデルを同様の目的で用いる。システムは、NewtonScriptオブジェクトの集合体として構築されるが、特定の関数は必要に応じC++で書く。APIがC++コード用に用意され、NewtonScriptオブジェクトの作成や操作ができる。こうしてC++の関数は、NewtonScriptの関数と同様、システムの他の部分とやりとりができる。

Newton C++ Tools

Newton C++ Toolsにより、NewtonScriptシステムの全体的枠組みに適したコードをC++で書くことができる。基本的な考えとしては、一群のC++ソースファイルをコンパイルして、ひとつの“ネィティブモジュール”としてリンクさせる。このネィティブモジュールは、プログラマーが定義する関数エントリーポイントを持つ一群のネィティブコードである。こうしてネィティブモジュールをNewton Toolkitプロジェクトに含めると、各々のエントリーポイントがNewtonScript関数オブジェクトとして利用可能となる。

ここまでくれば、あとはオブジェクトにエントリーポイントをどう割り当てるかにかかってくる。もっとも単純なテクニックは、単に関数を直接呼びだすことである。NewtonScriptのメソッドは、C++のエン

トリーポイントを“func”表現の代りにスロットに入れることで、C++に書き直すことができる。C++で書かれたクラスは、そのクラスのメソッドの“wrapper関数（折り返し関数）”を含むフレームを作成することにより、NewtonScriptへのインターフェースを持つことができる。

C++コードに与えられたAPIにより、NewtonScriptオブジェクトを作成・操作し、 NewtonScript 関数を呼び出し、NewtonScriptメッセージを送ることができる。NewtonScriptとC++のインターフェースは常にNewtonScriptオブジェクトが存在するが、C++データタイプへのまたC++データタイプからの変換のための“glue routines（糊付けルーチン）”を書くことは容易にできる。

Newton C++ Toolsは、“engine（エンジン）”コードに対して用いるのがベストである。エンジンコードとは、核となるデータ構造と動作を扱うかなり独立したアプリケーション部分を指す。多くのアプリケーションは、エンジン一個をユーザー・インターフェースで取り囲んだものである。 Newton C++ Toolsはエンジンに使用できる。（しかしエンジン内部で最高速が要求される部分のみに使用することが望ましい。）これに対し、NewtonScriptは、アプリケーションの残りのすべての部分に用いる。特に、C++コードの形で、完全なデバッグされたエンジンがすでにある場合には、便利である。

メモリなどNewtonデバイスのリソースは、既存のC++コードがもともと書かれたプラットフォームと比較して、限られているのが普通である。したがって、既存のC++コードを使用する際には、目標となるプラットフォームの制約条件内で動作可能か、注意深く確認しなければならない。

Newton C++ Toolsは、Apple社が、継続的に、よりオープンで、柔軟な開発プラットフォームをソフトウェアデベロッパに提供しようとしている最新の例である。以上述べたように、制約はあるにしても、多くのデベロッパが、Newton C++ Toolsを利用して、既存のCまたはC++で書かれたルーチンをNewtonプラットフォームに移植することができると考える。

NTJ

1ページからの続き

Newton OS 2.0搭載の新機種MessagePad 130をApple社が発表！

たりすることが容易にできる。ELバックライトは、電源スイッチを押すだけでONになり、タイムアウト機能は、ユーザーが環境設定を開いて設定できる。

サードパーティーのNewtonデベロッパは、必要があればバックライトAPIにアクセスでき、ソフトウェア内でバックライト制御を利用できる。三つのNewtonScriptコールが追加されバックライト制御を行う。

BackLightPresent ( )
ユニットがバックライトを備えていればtrueを戻す。

BackLightStatus ( )
バックライトがONならばtrueを戻す。

BackLight (RefArg onOrOffArg)
バックライトをONまたはOFFにする。以前のステートを戻す。

増設されたシステムメモリ

MessagePad 130は、512Kのシステムメモリが追加され、合計1MBのシステムメモリを持つ。この増設で、TCP/IPやワイヤレスLANなど通信ソリューションのパフォーマンスが向上した。システムメモリが追加されたことで、マルチタスキングサポート性能が向上した。システムメモリの追加に伴い、システムヒープも拡張された。

ヒープサイズの変更はApple Newtonベースのハードウェアの標準ではない。ヒープの追加は、環境の変化や限界条件の下で、より安定した動作をする通信アプリケーションの実現を目的としたものである。MessagePad 120は、MessagePad 130と同じ販売チャネルで従来からのヒープサイズで販売される。アプリケーションの設計開発にあたっては、このことを念頭においていただきたい。

耐久性の向上したノングレアスクリーン

MessagePad 130は、新しい、より耐久性のある、ノングレアスクリーンをそなえ、大きく異なる照明環境の下でも楽に情報の確認ができる。タブレットの耐久性が改善され、MessagePad 130は、ハードウェアとしてより丈夫で安定したモバイルコンピュータであり情報収集のツールとなった。

MessagePad 130の標準パッケージの内容

MessagePad 130には、電話記録、世界時計、公式ソフトなどを含む、各種の生産性を向上させるツールが組み込まれている。Newton Backup UtilityとシリアルケーブルがMac OSとWindowsベースのコンピュータ用として入っており、PC上でのバックアップおよびリストア、ならびにMessagePadへのインストールができる。

NTJ

Newtonで決め手のアプリケーションを創造する七つの方法

by ガイ カワサキ

編集室より

1995年のNewtonプラットフォーム・デベロッパーズ・カンファレンスで、ガイ カワサキは基調講演を行い、ガイのアドバイスの叡智に出席者は感銘を受けました。この講演内容を出版して欲しい、という読者からの度重なるリクエストが寄せられました。紙面の制約上全文を掲載することはできませんが、ガイ カワサキはその要旨を以下にまとめてくれました。七つのキーポイントは、Newtonアプリケーションの開発戦略の鍵となるものです。お楽しみを！

1) 過去は無視せよ。既存のソフトウェアを新しいプラットフォームに“移植”するのは“いくじなし”のやることだ。古いアプリケーションのタイプが古いプラットフォームで売れたから、新しいプラットフォームでも売れると考えるのは“愚かもの”だ。大うけするアプリケーションを作ろうとするなら、今まで歩いて来た道をはずれ、新たな道を切り開く必要がある。

2) 百花繚乱を求めよ。人が変わったアイデアを持ち込んだら耳をかたむけよ。（たとえば、Newtonベースのデバイスが博物館のツアーガイドの役を果たすなんてのを想像してみたらどうだ！）Appleは、別にマックの音楽ソフトの福音を説いたわけじゃない。音楽ソフトが発達し、それゆえに市場が大きくなったのは、何百という花が咲き乱れたからだ。

3) Appleの先を行け。あとから行くな。DTPはAppleが計画したものじゃない。Paul Brainerd (PageMaker)やJohn Warnock (PostScript)がいなかったら、Appleは今ごろ姿を消していたかもしれない。彼等がAppleをデスクトップ・パブリッシングに引っぱっていったのだ。自分の行きたいところに行け。そうすればAppleはついて行けるかも知れない。たとえAppleがついて行かなくても、わが道を行け。

4) 顧客を無視せよ。お客は、既存製品にたいする新機能の付加など、進化については大いに語れるが、革命的な製品をいかにつくりだすかについては語ることができない。もしわれわれが1984年にお客のいうことを聞いていたら、今ごろきっとApple XXIIをつくっているに違いない。ねらって、ブッ放せ。自分でたまをうつから、もうけも大きいのだ。

5) 自分で使いたいと思う製品をつくれ。WozがApple IとApple IIを設計したのは、自分で使いたかったからだ。どこかお高いところのマーケットスタディやフォーカスグループのデータを見てやったわけじゃない。もし自分が使いたい製品をつくれば、間違いなく一人はお客がいる、と確信できるわけだ。どんなにマーケットリサーチをやったって、こんな確信は持てるもんじゃない。

6) 資金を持ちすぎるな。金は足りない方が、ありすぎるより、問題が少ない。資源が不足している方が、賢く、ゲリラ的に、攻撃的に、動ける。金が沢山あると、カッコいいオフィス家具に便箋をそろえ、記者会見にシュリンプカクテルを山盛りにする、といった誘惑になる。

7) 兵は拙速を尊ぶ。こう私がいったと私は決してみとめないだろうが。Newtonのような製品の初期段階では（今だに初期だ）、製品は早く出してコンセプトをテストした方がいい。ベータサイトが要求する機能をみんなそろえた“完全バージョン”を待っている間に“立ち枯れ”してしまう可能性がある。主要機能を組み込んで出荷せよ。そして製品の妥当性をチェックせよ。そして迅速に改善せよ。

編集室より

ガイ カワサキはApple Fellowであり、その知性において伝説上の人である。

"How to Drive Your Competition Crazy"の著者。Forbesのコラムニストである。

覇権主義をストップさせせよう。福音主義者の仲間になろう。

E-Mailの宛先を＜macway-request@solutions.apple.com＞にして、どんなメッセージでもお送り下されば、自動応答いたします。アーカイブのサイトは：＜http://wais.sensei.com.au/searchform.html＞です。

NTJ

AppleのNewton デベロッパプログラムに関するお問い合わせ、またはお申し込みは下記へご連絡ください。

アップルコンピュータ株式会社

Newton Developer Support事務局

電　話：03-5334-2480

ＦＡＸ：03-5334-2781

email：jnewtondev@asia.apple.com

Letter From the Editors

Newtonデベロッパの皆様へ

今からおよそ一年半前、私たちはNewton Technology Journalの創刊号を世に送り出しました。これがどのように迎え入れられるのか、そして今後どうなっていくのか、希望と期待に胸は高鳴る一方でした。そしてこの一年半というもの、私たちの希望や期待が裏切られたことは一度もありませんでした。それどころか、Newton Technology Journalは、Newtonプログラミング技術を解説し、ドキュメンテーションやその他のサポートツールではカバーできないような、Newton OSの詳細な領域までカバーできる、優れたメディアであることが立証されたのです。このジャーナルは、AppleのNewtonシステムズ・グループからの最新ニュースとマーケティング情報をお届けするとともに、新製品を発表したり皆様を激励したりするフォーラムとしても利用されてきました。

Newton Technology Journalは役に立つという皆様からの反響には、スタッフ一同、感激しております。しかし、私たちは今のレベルでよしとしているわけではありません。これまで発行された通算7号には、Appleのエンジニア、トレーナー、プロダクト・マーケティング・マネージャー、そして独特なプログラミング技術のコツを得ている少数のデベロッパの方々が書いた、優れた記事が満載されています。今後もこれらの企画を続けていきたいと思っていますが、このジャーナルをさらに発展させ、Newtonデベロップメント・コミュニティのためのフォーラムにして、プログラミングのチップスやコツを共有したり、マーケティングの成功例を発表したり、あるいは追加のプログラミング情報を流すための場にもしたいと考えております。“ドクターLlamaに質問”のようなQ&A形式のコラムなどもおもしろいのではないでしょうか。皆様から質問を送っていただき、こちらが回答を提供するわけです。このジャーナルは、Newtonプラットフォームの優れたソリューションを追求する皆様のための雑誌です。改善してほしい点をお知らせくだされば、必ず対応させていただきます。また、おもしろかった記事、もっと続けてほしい記事についてもお知らせください。皆様のご要望に沿う内容をお届けしてまいります。

ご意見、ご要望をお知らせいただくのみならず、皆様ご自身にも記事を執筆していただきたいと考えております。経験豊かなNewtonのプログラマである皆様に、その専門技術をデベロップメント・コミュニティの仲間と分かち合っていただきたいのです。

今後もすばらしい記事をお届けしてまいります。皆様からのお便りをお待ちしております。

Apple Computer, Inc.
Newton Technology Journal編集部

Apple Developer Group

Newtonデベロッパプログラム

現在Newton Developer Support事務局では、日本におけますNewtonデベロッパサポートプログラムを再開するための準備を進めております。今回このような形でNewton Technology Journalの日本語版を提供させて頂きますとともに、デベロッパの皆様、特にNewtonのデベロッパサポートをご希望の皆様へのご挨拶とさせて頂きたいと存じます。なお、プログラムのサポート内容、開始時期、費用など詳細につきましては近日中にご案内差し上げたいと存じます。少しでもデベロッパの皆様のお役に立てますようなプログラムにして参りたいと存じますので、何卒宜しくお願い申し上げます。

なお、お問い合わせなどございましたらお手数ですが、下記連絡先まで御連絡くださいますようお願い致します。

AppleのNewton デベロッパプログラムに関するお問い合わせ、
またはお申し込みは下記へご連絡ください。

アップルコンピュータ株式会社
Newton Developer Support事務局
電　話：03-5334-2480
ＦＡＸ：03-5334-2781
email: jnewtondev@asia.apple.com