モバイルコンピューティングと
ワ イ ヤ レ ス 通 信 22−13
（２００２．１０．１８）

# 携帯電話の SW 更新に関する検討

清原　良三† 栗原　まり子† 高橋　清‡ 橘高　大造†

近年の i-mode をはじめとする携帯電話のネットワーク接続サービスの開始により、携帯電話移動機上に搭載するソフトウエアの規模は急激に拡大している。そのため障害のない状態で携帯電話を市場に出すことが困難な状態になりつつある。そこで、市場に出した後にも携帯電話移動機ソフトウエアの更新をする必要性が想定される。

ソフトウエアの更新においては携帯電話に送りこむデータ量によって左右される更新時間が問題となる。本論文ではソフトウエアの更新を前提とした携帯電話移動機ソフトウエアの構造に関して検討を行うとともに、方式提案およびその評価に関して報告する。

# A Study of Updating Software on Mobile Phones

Ryozo Kiyohara† Mariko Kurihara† Kiyoshi Takahashi‡ Taizo Kittaka†

Due to increasing services of cellular phones(e.g. i-mode), it is difficult to release bug-free cellular phones. We have developed a technology for updating cellular phones' software that requires less time. This updating time depends on data size which is delta between old version and new one. In this paper, we discuss and evaluate about the software architecture and development environments that reduce delta size.

## 1. はじめに

最近、i-mode サービスをはじめとしてネットワーク接続可能な携帯電話移動機（以下、「移動機」と表現する）が普及してきた。これらにはカメラ、赤外線 I/F および JavaVM まで搭載している機種もある。そのため移動機上のソフトウエアの規模が急激に大きくなっている。しかしながら開発サイクルの問題からも十分な試験期間を設けることができず、障害のない状態での出荷が困難になっている。そこで、出荷後に移動機上に搭載したソフトウエアの更新の必要性が高まっている。そのためのソリューションの提供も始まりつつある。[1][2][3]

ソフトウエアの更新は、移動機を回収し、工場でソフトウエアを書き直す、またはショップのパソコンを利用して書き直すことにより実

†三菱電機（株）情報技術総合研究所
Mitsubishi Electric Corporation Information Technology R & D Center
‡三菱電機（株）モバイルターミナル製作所
Mitsubishi Electric Corporation Mobile Terminal Center
kiyohara@isl.melco.co.jp

現することがまずは想定される。しかし、移動機上の全ソフトウエアを更新する為には、移動機へのデータの転送速度から考えても多大な時間を要する。例えば、12M バイトのプログラムサイズとすれば、256Kbps で送信可能としても、単純計算で約６分強はかかることになる。これにフラッシュ ROM の消去の時間などの様々な処理を考えるとさらに時間がかかることになる。

この書換え時間は工場で大量に処理する場合は、その処理能力に大きく影響し、ショップの店頭でパソコンを利用して書き換える場合は、持ち込んだユーザの待ち時間に影響を与えることになるため、できる限り高速である必要がある。そのために、全プログラムを送信するのではなく、前のプログラムとの差分情報のみを送信する方法が考えられる。しかしながら、ソフトウエアの作りによっては差分の大きさが大きく変わるため、始めからソフトウエアの更新を想定し、差分が小さくなるような開発環境を使う必要がある。

本報告では、移動機のソフトウエア更新の課題の中でも、ソフトウエア更新時の旧版と新版とのコードの差分を小さくするための手法に関して提案するとともに、その方式に関して評価を行い、その有効性を示す。

## 2. 携帯電話 SW の構造の検討

### 2.1. 基本的構造

移動機のソフトウエアの構造は基本的には一般的な組み込みシステムのソフトウエア構造と同じである。OS、ドライバ、ミドルウエア、アプリケーションなどがあり、これらがフラッシュ ROM 上に書き込まれている。またメモリはコストの関係からも高速性を要求する部分と容量を要求する部分に分けた構成をすることが多い。これらのメモリをユーザのデータを置くユーザ領域、ユーザの設定情報などを置くシステム管理のユーザ領域およびシステム領域に分けることができる。ここでソフトウエアの更新の対象となるのは、システム領域のフラッシュ ROM 上の実行用のコードとデータである。

移動機では PC 上のソフトウエアとは違い新しいソフトウエアをダイナミックに追加したり置き換えたりする必要性は今まではなかった（Java のみ例外であった）。そのため、PC のソフトウエアにあるような DLL の機能を使っていない場合が多い。

ソフトウエアの更新のことを考えると DLL をローディング前の状態で保持し、ロードして利用する手法の方が適している。しかしながら、メモリ効率は悪く、その実現のためには高価な RAM が必要になり、移動機のコストを抑えるという観点からは望ましくない。そこで DLL など利用せずに、リンク・ローディング済みのコードに対してソフトウエア更新するための方式に関しての検討結果を報告する。

### 2.2. 修正の局所化の検討

本節では局所的な修正が局所にだけ影響する方式に関して検討する。

**（1）パッチ**

図１に示すようにバイナリにパッチをあて、

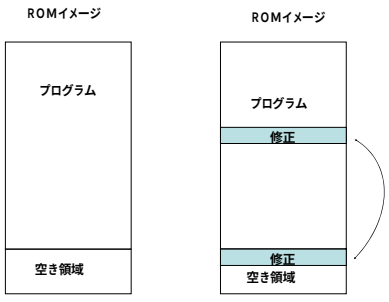


図１　バイナリパッチ

修正コードを最後の領域などに置く手法は一般に使われている方式である。修正した場所以外には影響を与えずに有効な手法であるが、ソースとの対応関係で管理や、修正後の出荷版とのバイナリイメージが異なる場合があり、ソースおよびバイナリの管理の観点から望ましい方法ではない。

**（２）関数間の参照関係をジャンプベクタテーブル経由にする方式**

コンパイラ、リンカによっては関数間の参照においてジャンプベクタテーブルを利用する方式や、空間に予備の領域を空けてリンクするものがある。これは図２に示すように、コードが追加になっても他のモジュールの位置に影響を与える可能性が少なくなり、関数の位置がずれてもジャンプベクタテーブルを利用することにより、関数のエントリは必ず位置固定となり変わらないため局所にしか影響を与えないという点で有効である。

**（３）共通データ**

参照されるデータに関しても場所が移動になればそのアドレスが変わるため、局所の修正にもかかわらず広範囲に影響が及ぶことがある。しかし、データに関しては経験的にも不具合の修正を入れる可能性は少ないと考えられる。そこでアクセスの高速性を要求するデータは上の最初の固定位置とすることで対処する。修正の可能が高く、アクセスの高速性が要求されないデータはＡＰＩを経由してアクセスするといった方法により解決するのが有効である。

## 2.3. モジュール分割方式

前記バイナリパッチ方式とジャンプベクタテーブル方式を比較した場合、ソフトウエアの更新の目的がバグ修正など不具合発生時を考慮したためであるという観点から更にソース管理など複雑にし、不具合の可能性を大きくすることがないようにするため、パッチ方式を採用せずに、ジャンプベクタテーブル方式を採用することとした。ただし、ファイル単位の外部参照宣言などを利用し、すべてのリファレンス関係にこのようなジャンプベクタテーブルを導入すると爆発的にメモリが必要になる。そのため、ある程度の塊をモジュールとし、このモ

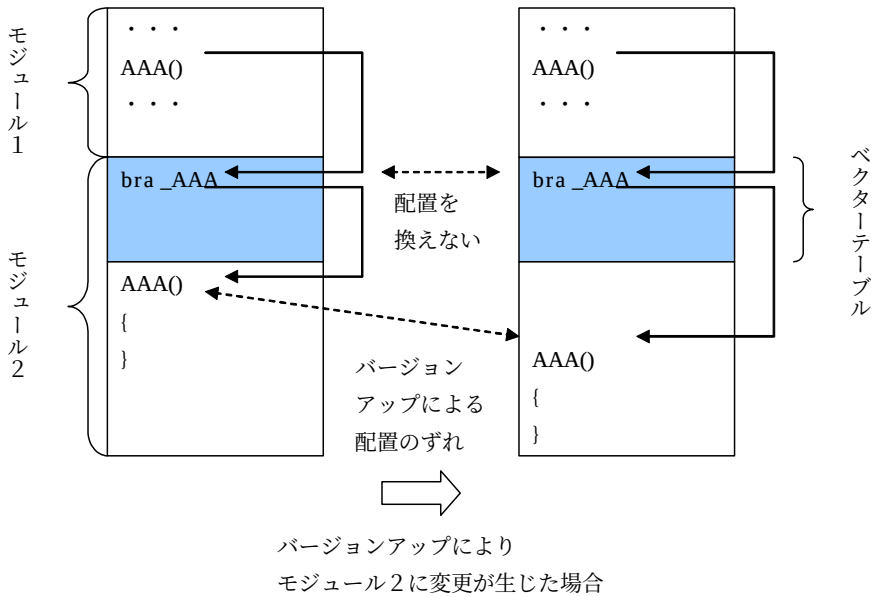


図２ジャンプベクタテーブル方式

ジュール間での参照に限りジャンプベクタテーブルを利用する方式とした。このようにすることによりジャンプベクタテーブルの巨大化を防ぐことができると考える。

## 3. 実現方式の検討

### 3.1. エリア

フラッシュ ROM 上のソフトウエアは移動機のコストを抑えるために、最小限のコストでオーバラップしながら実行するように様々なメモリ空間にコピーされて実行される部分がある。そこで、エリアという概念を設ける。図３に示すようにアクセスの速い内蔵のＲＡＭエリア、ＣＰＵの外部のＲＡＭエリアで揮発性のものあるいは不揮発性のものがあると想定される。また、ＲＡＭの内容はＲＯＭからコピーされ、これはオーバラップを許すことにより効率的に利用できる。この他にデータキャッシュが有効なエリアや命令キャッシュが有効なエリアが限られる場合などもあり、更に実際にはより細かく分けられることになる。モジュール分割はこれらを意識して行う必要がある。

### 3.2. リンカースクリプトでの解決

モジュール分割は、本来参照関係を考え、最も弱い関係となるように分割するべきである。しかしながら、実装メモリを考えると静的に参照関係を解析するのみで判断できるが、実行性能を考えると動的にその参照関係を見て判断する必要がある。そのため一概にはなんとも言えない。そこで通常利用するリンカースクリプトを拡張し、モジュール分割宣言を導入し、開発中、あるいは開発後にもこのリンカースクリプトの書き方を換えるだけでモジュール構成を変えることができるようにする。

ジャンプベクタテーブルはコンパイル･リンク手順の中で自動的に挿入することができる。また、ICE を利用したデバッグ時であるが、開発者がジャンプ系命令の場合はジャンプベ

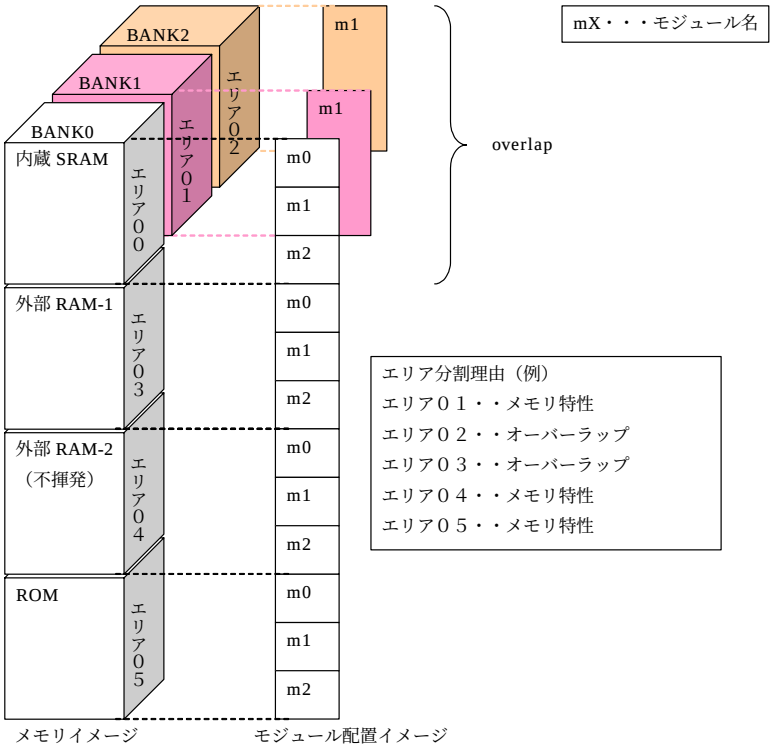


図３エリア概念図

単一リンクパラメータファイル

エリア情報

```
;%%%AREA_NUM 2
;%%%AREA00 00000000 00FFFFFF 8 RAM
;%%%AREA01 01000000 FEFEFEFE 9 ROM
```

モジュール情報

```
;%%%MODL_NUM 2
;%%%MODL00 m0
;%%%ADDR00 m0 A 00111111 A 00222222 A 00333333
;%%%ADDR01 m0 A 04444444 A 05555555 A 06666666
;%%%MODL01 m1
;%%%ADDR00 m1 N ******** A 00888888 A 00999999
;%%%ADDR01 m1 A 0AAAAAAA A 0BBBBBBB A 0CCCCCCC

%PATH_VECT vect\
%PATH_GSX  gsx\

/ADDR = 00222222
;%%%STR m0 00
;この間が m0 にコピーされ、ｴﾘｱ 00 のアドレスが記述される
/SECT = ・・・
;%%%END m0 00
```

図４リンカースクリプトの例

クタテーブルを経由する場合があることを認識していればなんら問題がないと考える。またエリアの宣言もリンカースクリプト内に記載することにより、速度チューニングの結果でのメモリ配置の変更もリンカースクリプトの変更だけで行うことができる。図４にエリア宣言などリンカースクリプトの例を示す。

### 3.3. コンパイル・リンク手順

モジュールやエリアという概念をリンカースクリプトに導入したが、一般のコンパイラ/リンカではこれらの表現を解釈する機能を持たない。また、ユーザの意図したモジュール分割にする必要性から、これらの宣言を解釈した上でモジュールに分割し、ジャンプベクタテーブルを生成するとともに、分割リンクを導入し、解決できないシンボルに関しては別途定義済みのシンボル情報のみを入力させる必要がある。

これらを実現するための基本機能は gnu を代表とするリンカには備わっているため、実際にはモジュール分割に伴うジャンプベクタテーブルとそのシンボル情報を抽出し、入力シンボルとする。図５にコンパイル・リンク手順を示す。

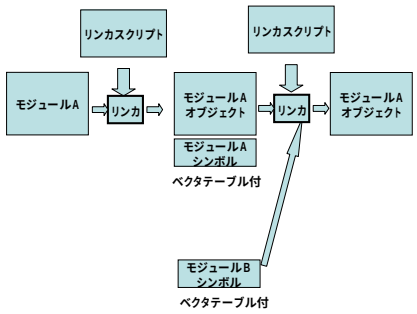


図５コンパイル・リンク手順

## 4. 実装

### 4.1. モジュール分割

ジャンプベクタテーブル方式についても以下の２つの実装が考えられる。

(1) テーブルにアドレス情報を入れておき、実際にジャンプするときに、このテーブルの固定位置をレジスタに読み込んでジャンプする方式

(2) テーブルにはフラグの状態などに影響を与えないジャンプ命令を直接入れておき、実際にはこのテーブルに直接ジャンプし、２段ジャンプで実装する方式

ここで現状主流として利用されている32bitCPU を想定し、(1)の方式の方がメモリ使用率、命令数とも(2)より大きくなる場合が多い。しかしながらジャンプが１回であるためキ

```
(1) レジスタにアドレスを読み込んでJumpする方式例
        seth テーブルアドレス      32bit 命令
    /* テーブルアドレスの上位16bitをレジスタにロード */
        setl テーブルアドレス      32bit 命令
    /* テーブルアドレスの下位16bitをレジスタにロード */
        jl   ジャンプ &リンク      16 bit 命令
    /* レジスタの内容で戻り先をレジスタに入れてジャンプ */

(2) ジャンプテーブル経由でJump する方式
        bl   テーブルアドレス      32bit命令
    /* 即値で指定されたPC相対アドレスに戻り先を
                    レジスタに入れてジャンプ */

テーブル: bra  ジャンプ & リンク (相対) 32bit命令
    /* 即値で指定されたPC相対アドレスにジャンプ */
```

図６　関数参照命令の例

ャッシュやパイプライン処理を考えても速度的には(1)の方が良いと考えるが、後述するメモリの増加率の関係で(2)の方式の採用とした。

### 4.2. モジュール構成

モジュール構成はできる限りモジュール間の関係が少ないことが望ましい。この最適な関係を静的、動的に求めて解析した上で決めるのが最も望ましいが、ソフトウエアの不具合修正後に変わる可能性もある。そこで

(1)モジュール間の関連が少ないこと

(2)修正が全域に波及しないこと

という観点から、UI や通信といった単位で全体を１０個から 15 個の程度のモジュール構成とすることとした。

## 5. 評価

ある機種の移動機のソフトウエアに対して実際に適用した結果と評価を述べる。

### 5.1. メモリ効率

メモリ効率はモジュール分割した場合の外部参照関係がどの程度あったかで決まる。表１にモジュール分割をした結果を示す。モジュール間参照のＡＰＩ数が合計で約 3000 であるためジャンプベクタテーブルとしては 32bit x 3000 で約 12KB のジャンプベクタテーブルが必要となる。これはこの機種のＲＯＭ容量である 12MＢに対して約１％のメモリ増加であり、許容範囲ではないかと考える。モジュールの分割数を増やせばこの割合は増加する。ファイルをモジュールと考えた場合は、2 桁以上変わるためメモリの増加量は誤差ではなくなるため１０個程度でのモジュール分割は有効である。また予備の空間として各モジュールにフラッシュＲＯＭの消去の単位分程度の余分なスペースをとったとして、フラッシュＲＯＭの消去の単位が 64KB であるなら、64 x 12 = 768KB であり、6%程度となる。平均的に考えれば 3%

表１　モジュール分割結果

| | |
|---|---|
| ソフトウエアサイズ | 約 12MB |
| モジュール分割数 | 12 |
| モジュール間参照 APＩ定義数 | 約 3,000 |
| モジュール間参照呼出し数 | 約 40,000 |
| モジュール間データ参照 | 約 300 |
| モジュール間データ参照におけるデータサイズ | 約 600KB |

程度であり、過去機種からの経緯から安定したモジュールの余裕は少なくするなどすれば更に減らすことができるため、問題ない範囲と考える。

### 5.2. 実行時性能への影響

メモリの効率に関しては静的な解析の結果で判断できるが、実行時性能への影響はモジュール間の参照が実行時間に比べてどの程度あるかを動的に知る必要がある。ここで、実行時の速度で最も重要だと考える 2 点に関して実測を試みた。

(1)　キー入力の反応時間

キー入力は、基本的にはある程度のチャタリングと長押し判定との両者の判断を経て各種アプリケーションに伝わる。（もちろん押下イベントのみが伝わる場合もある）ここではキー入力の割り込みがあってからアプリへ配送されるまでの時間を測定した。その結果処理時間のオーバヘッドは 2%程度であった。これはチャタリング判定や長押し判定の時間に比べて十分短くまったくキー入力における速度に影響を与えていないことがわかる。

(2) Java の実行速度

携帯電話のキー以外の部分で速度が気になるのは描画速度や、画面の切り替わり速度などである。これらの機能を最も使うものとし

表２ KVMMark の性能劣化率

| 項目 | 劣化率(%) |
|---|---|
| Calc | 0.3 |
| Loop | 0.3 |
| Method | 0.5 |
| Scratch | 1.5 |
| Panel | 2.8 |
| Graphic | 0.7 |
| Image | 0.6 |
| 合計時間での劣化率 | 1.2 |

て携帯電話向き Java がある。本影響に関して Java を搭載している機種を利用して測定してみた（表２）。測定にはベンチマークアプリである KVMMark[4]を利用した。その結果実行時間に与える影響は 0.3% - 2.8% の間であることがわかった。平均としては 1.2%でありメモリ効率の観点とほぼ同程度のオーバヘッドということができる。

以上の観点から速度的な観点のオーバヘッドは許容できる範囲内であると考える。

### 5.3. 差分情報量

実際の差分はどの程度になるかを示す。差分の評価の方法としてここではフラッシュＲＯＭの変更点を見るために、フラッシュＲＯＭのイレースブロックの単位で見て何ブロックに変更があったかということで評価した。この値が小さければ良い。表３には、100 行程度の関数を一つ追加した場合、即ち追加した部分とその関数を呼ぶ２つのモジュールに修正が入った場合において、モジュール分割前、モジュール分割したがジャンプベクタテーブルを使わない場合、モジュール分割およびジャンプベクタテーブルを利用した場合と示す。何もしない場合に比べてモジュール分割とベクタテーブルにより、約半分の領域の修正で済む。この場合は対象モジュールも大きかったため半分程度であるが、小さいモジュールに修正を入れた場合はさらに小さくなる。

表３　試験データによる差分量の比較

| | |
|---|---|
| 書換え対象ブロック数 | 194 |
| モジュール分割なしの場合 | 68 |
| モジュール分割実施した場合 | 59 |
| モジュール分割およびジャンプベクタテーブルを導入した場合 | 37 |
| 修正対象となったモジュールの全ブロック数 | 39 |

さらに、モジュールを粗く分割しているためにプログラムの位置のずれによる影響も受けており、モジュール内では全域に修正が渡っている。即ち、位置独立なコードでも位置がずれていればフラッシュＲＯＭ上では書換えが必要である。そこで、バイナリ差分を位置に独立な形で抽出してみた。即ち、単純に位置がずれているだけでのデータを調べてみた。図７に示すような位置がずれるデータをコピーという形で表現するために、コピー元のアドレスとコピー先のアドレスおよびデータサイズを付与した差分データサイズと、全くずれではなく異なっていた部分のデータサイズとの合計をし

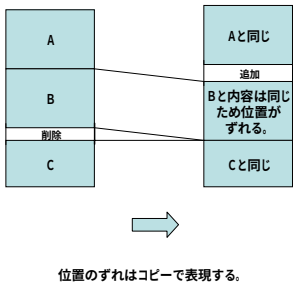


図７差分の表現方法

てみると、約３００Ｋバイト程度であった。即ち、実際には２MのフラッシュＲＯMの書換えが必要であるが、３００Ｋバイトあれば十分差分データを表現できることがわかった。これはずれているコードは単にずれていると表現するだけの修正をするだけで差分情報量という観点から十分効果があることを示している。

### 5.4. 開発環境

ユーザにとって開発環境が変わると開発効率が落ちてしまう。この観点から評価する場合は次の 2 点であると考える。

#### 5.4.1. 開発手順の変更

開発手順に関してはコンパイル･リンク手順に一部ツールを使った手順が入るため使いにくくなると考える。しかしながら 12MB を超える規模のソフトウエアとなると大人数での開発であり、正式なリリースのためのコンパイル・リンクをする人は限られると考える。限られた慣れた人のみが実行する手順が複雑になる程度であり、問題ないと考える。

#### 5.4.2. デバッグ手順の変更

モジュール間のジャンプにおいてはベクタテーブルを経由する。これはソースコード上にないもので、ソースコードデバッグを行う際に、まったく予想しない部位にジャンプしてしまうが、これは本開発環境の概念を理解していれば問題ない範囲と考える。また、シンボルの問題で、モジュール間でジャンプした場合に ICE 等が利用するシンボルファイルを切り替える必要が生じる。ICE によっては厄介な操作が必要になる場合もあれば、マクロ定義などで自動的に切り替えることも可能な場合もある。

ただし、ここでは多くの人は自分の担当するモジュール内のみでデバッグするのであって、モジュール間に渡るデバッグをする人は限られていると考え、十分許容されるものと考える。

## 6. おわりに

本研究では、携帯電話移動機のソフトウエア更新を旧版と新版の差分のみを送ることによって実現する方式において課題となる差分が小さくなる開発環境およびソフトウエア構造に関して検討を行い、実際に試作を行うことにより移動機を利用して評価を行った。その結果若干のオーバヘッドがメモリ効率および実行時性能にあるが 1%～3%程度であり十分許容範囲内に収まることを示した。

また差分情報量に関してもモジュール分割などを実施しない場合に比べて十分効果があることを示した。開発手順や、デバッグ環境においても若干の変更があるものの効果に比べれば問題は小さいと考えられることから、本方式で、十分差分の小さくなる携帯電話のソフトウエア構造と開発環境を実現したと考える。

今後の課題としては次の点を検討していく予定である。

(1) 位置に独立なコード部の位置のみがずれている場合などを差分の表現方法を工夫するなどして更に差分を小さくするための差分抽出方法と差分表現方法の検討
(2) 有限でフラッシュＲＯMに比べて極端に小さいＲＡMを利用したソフトウエア更新機能の端末上での実装。
(3) 更新失敗時のリカバリに対する検討
(4) 更新失敗を防ぐためのセキュリティ機能の検討

【参考文献】

[1] http://www.doongo.com
[2] http://www.bitfone.com
[3] 日経エレクトロニクス,「携帯電話機のバグをユーザの手元で修正へ」 2002 年 3 月,PP.22-23
[4]http://www.seckey.net/iappli/KVMMark.html