複合的なセンシング処理を端末ハードウェアの構成に合わせて 最小の電力で行うセンシングミドルウェア

佐藤卓也[†1] 長谷川英司[†1] 中尾学[†1] 上和田徹[†1] 松本達郎[†1]

概要：現在のスマートフォンには様々なセンサが搭載されており，センサを使って人の行動や周辺の状況を継続的にとらえてサービスを提供する事例が増えている．しかし，行動や状況を継続的にとらえるためにはセンシングを継続的にする必要があり，消費電力が増える要因となっている．これに対し，センシング処理を省電力なコプロセッサで行い，条件を満たした時だけ CPU を wake することで消費電力を抑える技術が実現されている．しかし，複数のセンシング結果を活用する場合，それぞれのセンサでイベントが生じるたびに複数のセンシング結果が条件を満たしたかを判定するために消費電力の高い CPU が wake されてしまう．そこで，複数のセンシング結果の判定を省電力なコプロセッサで行うセンシングサブシステムを提案する．また，アプリ開発者がセンシングサブシステムの有無といった端末のハードウェア構成を意識しなくても，構成に合わせて最小の電力でセンシング処理を行う API とミドルウェアを提案する．そして，これらの提案手法を実装し，消費電力を評価した．この消費電力をもとに業務支援での利用を想定した試算を行い，今回の提案手法により 4.1%の消費電力が削減されることを確認した．

Sensing middleware for performing the complex sensing process by minimum power in accordance with adjusted to terminal hardware component

TAKUYA SATO[†1] EIJI HASEGAWA[†1] MANABU NAKAO[†1]
TORU KAMIWADA[†1] TATSURO MATSUMOTO[†1]

1. はじめに

現在のスマートフォンには様々なセンサが搭載されており，加速度センサによる画面回転，GPS によるナビをはじめとする様々な便利な機能が提供されている．また，ユーザーの場所に応じた情報を提供する Google Now[1]や，ユーザーの移動経路を記録する My Tracks[2]といった，センサを使って人の行動や周辺の状況を継続的にとらえてサービスを提供する事例が増えている．

しかし，行動や状況を継続的にとらえるためにはセンシングを継続的にする必要があり，消費電力が増える要因となる．例えば，ユーザーの移動距離や歩数を測る ARGUS というフィットネスアプリを iPhone5 以前の機種で用いると, 1 日に 20～30%程度のバッテリーを消費してしまう[3]. これは CPU と通信の高速化や大画面化に伴い電池の持ちが大きな課題になっているスマートフォンでは看過できない問題である．

2. センシング省電力化の従来技術

人の行動や状況を継続的にとらえる上で，センサデータの処理を行うために消費電力の大きな CPU を動かし続けると消費電力が大きくなる．これに対し，継続的に行う単純なセンサデータの処理を，性能は低いものの消費電力が小さいコプロセッサにさせることで消費電力を抑えることが考えられ，最近のスマートフォンには，CPU より低消費電力でセンサデータの継続的な処理をできるコプロセッサが搭載されるようになってきている．その代表例として，Apple M7[4]や，Motorola X8 の Contextual Computing Processor[5]があげられ，例えば，コプロセッサが加速度センサからの継続的なデータをもとに歩行状態を判定し，変化があった時のみアプリに通知をすることで，CPU の動作を減らすことができる．

3. 従来技術における課題

行動や状況を継続的にとらえて活用するサービスにおいては，１つのセンサだけではユーザーの状況を絞り込むことができなく，複数のセンサを組み合わせて使う場合が生じうる．例えば，店で人が立ち止まって手に取った物をモニタリングするようなサービスを実現する場合には，歩行センサと物検知センサを同時に用いてユーザーの行動や状況を捉え続ける必要がある．

このような複数のセンサを組み合わせた判定では，一方のセンシング結果によって，もう一方のセンシング結果の要・不要が変化することがあるが，この判定を CPU で行う

†1 (株)富士通研究所

とセンシング結果が不要な場合であってもその判定のためにCPUが動作してしまう場合がある．図1では，複数のセンサからのイベント発生状況の例を時系列で示す．センサ1とセンサ2の軸にある縦線は，センサイベントが発生したタイミングを示す．ここで，センサ1と2でほぼ同時にセンサイベントが発生した時のみを，アプリやサービスを実行するトリガとするケースを考える．Aの期間では両方のセンサから情報が上がってきているので，センサ2のイベントはアプリやサービスを実行するトリガとなるが，Bの期間ではセンサ2からしか情報が上がってきていないので，センサ2のイベントはアプリやサービスを実行するトリガにはならない．しかし，Bの期間においてもセンサ2でイベントが発生するたびにCPUがwakeされ，トリガが不要であるという判定処理が行われる．例えば，先ほどの店で人が立ち止まって手に取った物をモニタリングするようなサービスでは，人が停止している間のみ，物の検知の判定をすれば十分であるが，人が動いている間まで物検知センサからのイベントを判定するためにCPUが動く．このようにアプリで使わないイベントが発生した時でもCPUが動作してしまい，電力を消費することが課題である．

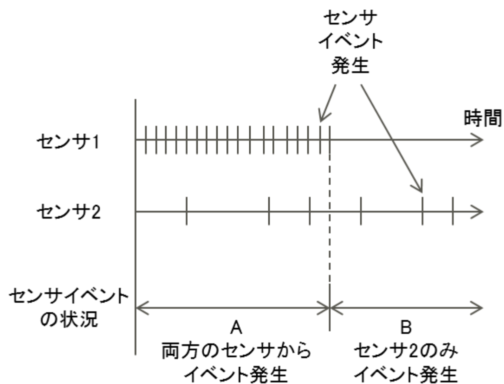

図 1　複数のセンサからのイベント発生状況の例

4. 提案手法

複数のセンサからのイベントの組合せ判定を消費電力の大きなCPUで行うことによる電力課題を解決するため，CPUより省電力なコプロセッサでアプリに必要なイベントであるか判定し，必要なイベントのみアプリに通知する仕組みを提案する．この仕組みでは，どのようなセンシング処理が必要かをアプリに指定させて，コプロセッサでアプリへの通知が必要か否かの判定を含めたセンシング処理を行い，アプリの指定したセンシング結果が得られるまでCPUをスリープ状態にする．図1の時と同じセンサイベントが発生した場合を例にとると，Bの期間はコプロセッサでセンサ2のイベントを受け取るが，センサ1のイベントがないのでCPUはwakeしない．このようにすることで，CPUの動作頻度を減らすことができる．

4.1 複数のセンシング結果の判定を行うサブシステム

アプリへのイベント通知判定をコプロセッサで行う仕組みとして，コプロセッサが複数のセンサを持つことで，CPUと独立してセンサイベントの処理ができる複合センシングサブシステムを提案する．このサブシステムは，アプリから通知条件を受け付けて，その条件を満たした時だけアプリに通知する機能を持つ．

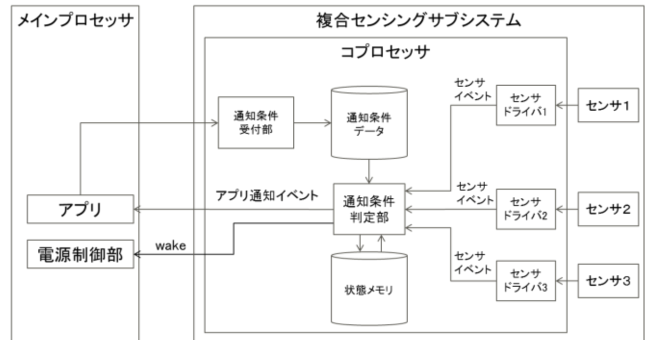

図 2　複合センシングサブシステムを含む
システムの全体構成

図2は複合センシングサブシステムを含むシステム全体の構成図を示す．このサブシステムでは，アプリから通知条件を受け付けて記憶する．そして，センサドライバからイベントが発生するたびにそのイベントと通知条件を照合する．この照合の結果が，指定された通知条件に合致した場合のみCPUにwake信号を送り，アプリに通知をする．複数のセンサからのイベントは時間のずれが生じることがあるので，過去のセンシング結果を保存するために状態メモリという構成要素を持たせておくことで，イベントの時間差を許容できるようにする．

4.2 複合的なセンシング処理を行うミドルウェア

アプリからセンシングをするには，現在スマートフォンでよく使われているセンシングAPI[6][7]を使うことになる．この場合，アプリがそれぞれのセンサから値を得ることになるため，複合センシングサブシステムの通知条件判定機能は利用できず，アプリがセンサイベントに対する判定を行う必要がある．そうすると，アプリが必要とするイベントのみを通知させることによる省電力化ができないため，このサブシステムに対応したAPIが必要となる．

ここで，複合センシングサブシステムを新たな種類のセンサと定義してAPIを設け，アプリから通知条件を指定できるようにする方法が考えられる．この方法を用いれば，アプリから指定された条件を満たした時のみCPUをwakeしてアプリに通知することが可能となる．

従来のAPIを用いた場合と通知条件判定をする複合センシングサブシステム用のAPIを用いた場合にセンサからイベント通知がされる様子を図3(a),(b)に示す．アプリが使用するセンサを指定する従来のAPIではそれぞれのセンサか

らアプリにイベント通知が行われるのに対し，このサブシステム用の API では，アプリから指定された条件を満たした時のみアプリに通知が行われる．

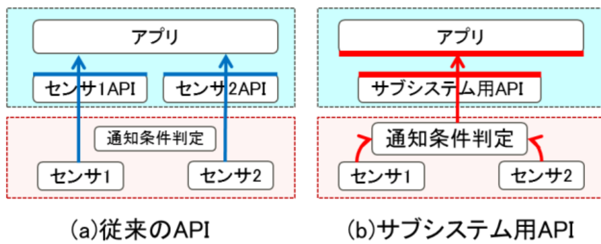

図 3 従来の API と複合センシングサブシステム用 API を使用した場合におけるセンサからの通知

しかし，このサブシステム用の API は対象とするサブシステムを搭載した端末でしか動作しない．そのため，複合センシングサブシステムを搭載しない端末でアプリを動作させるには，従来の API を用いてセンサごとにイベントを受け取り，アプリがイベントに対する判定をする必要がある．したがって，複合センシングサブシステムを活用して省電力化することと，端末によらずアプリが動作することを両立させるためには，アプリ開発者は，端末の構成を意識して構成に応じた異なる API を用いなければならない．

そこで，この課題を解決するために，アプリが使用するセンサを指定してセンサデータを得る従来の API の枠組みを変えて，アプリが取得したいセンシング結果を指定する API を提案する．

図 4 に従来の API によるセンシングの仕組みを示す．この API ではアプリがセンサを直接指定して値を取得する．ゆえに各センサイベントが発生するごとに CPU が wake される．これに対し，提案する API の仕組みを図 5 に示す．この仕組みではアプリがどのようなイベントを必要としているかを指定する．その指定はセンサドライバではなくセンシングミドルウェアが受け付ける構造をとる．そして，センシングミドルウェアが受け付けた通知条件に基づいて各センサにセンシング処理を実行させる．

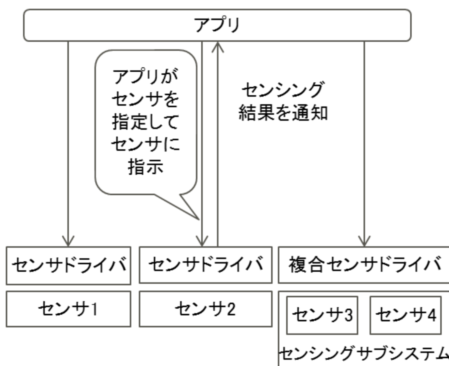

図 4 従来の API によるセンシングの仕組み

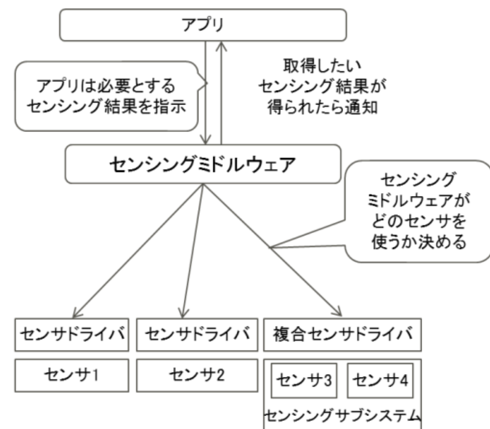

図 5 アプリからの取得したいセンシング結果の指示を受けてセンシング処理を行う仕組み

4.2.1 センシング処理の指定

この API においては，アプリから条件と処理内容を式 1 に示すような JavaScript 風の言語で指定する．この通知条件を指定するイベント判定文は，式 1 の condition が示す条件文と statement が示す処理文から構成される．条件文はセンサイベント情報，状態変数，定数を項に持つ．センサイベント情報はセンサから上がってきたイベント情報を保持する項であり，センサの種類とイベントの内容で構成される．状態変数とは，4.1 で述べた状態メモリに記憶させた変数を指す項である．定数は数値または文字列のリテラルである．これらの項に対して，比較演算や論理演算などの演算を記述することができる．処理文にはアプリへの通知や，状態変数への更新を記述する．以上をもって複数のセンサでほぼ同時にイベントが発生した時のみアプリやサービスにトリガをかけることを記述して指定できる．

if (condition)statement　　　　（式 1）

通知条件の記述例を図 6 に示す．最初の 2 行は状態変数を定義する．それ以下の 6 行が式 1 に示すイベント判定文である．センサイベント情報は，[センシング種別]::signal[イベントのフィールド名]で記述し，この形式でない変数はすべて状態変数を意味する．センサイベント情報の記述例を挙げると，センシング種別が場所センシングで type というフィールドを参照する場合は place::signal[“type”]と記述する．このセンサイベント情報に対して条件判定を行うには他の項と比較演算や論理演算などの演算を行う．例えば，place::signal[“type”]==”enter”という条件式は，場所センシングで種別が enter であるという条件を意味する．さらに，place::signal[“label”] ==”room”は，場所センシングでイベントに含まれる label という情報が，room であるという条件を意味する．すなわち，図 6 のイベント判定文の 1 行目は，場所センサで room に入ったというセンサイベントが発生したことを検知した時という意味になる．この時の処理文が，room=1 であり，状態変数 room に 1 を代入するという

```
var int room;
var int projector;
if((place::signal["type"] == "enter") && (place::signal["label"] == "room"))room=1;
if((place::signal["type"] == "enter") && (place::signal["label"] == "room") && (projector==1))notify();
if((place::signal["type"] == "exit") && (place::signal["label"] == "room"))room=0;
if((object::signal["type"] == "detect") && (object::signal["label"] == "projector"))projector=1;
if((object::signal["type"] == "detect") && (object::signal["label"] == "projector") && (room==1))notify();
if((object::signal["type"] == "lost") && (object::signal["label"] == "projector"))projector=0;
```

図 6　通知条件の記述例

処理である．まとめると，1 行目は場所センサで room に入ったことを検知したら状態変数 room に 1 を代入するという記述である．また，2 行目の処理文 notify()はアプリへ通知を行うことを意味する．すなわち 2 行目は，場所センサで room に入ったことを検知したのに合わせて，状態変数 projector が 1 であるときに，アプリへ通知するという意味になる.このようなイベント判定文を複数記述することで，複数のセンシング結果を組み合わせた判定を記述する．

4.2.2 センシングミドルウェアの動作

センシングミドルウェアはアプリから通知条件を受け付けて，その通知条件を満たした時にアプリに通知する．この時のセンシングミドルウェアの動作を図 7 に示す．センシングミドルウェアは，アプリからの通知条件の指定を受けて以下の処理を行う．

1）センシングミドルウェアが，アプリから通知条件を受け付ける．

2）センシングミドルウェアは，端末の持つすべてのセンサドライバに対し,指定された通知条件のうちどこまでを処理できるか問合せる．

3）問い合わせを受けた各センサドライバは，自らのセンサの処理能力をセンシングミドルウェアに回答する．この時，複合センサドライバは個々のセンサの能力に加えて,通知条件判定ができることも回答する.

4）各センサからの回答を得たセンシングミドルウェアは,アプリから指定された通知条件の多くを任せられるセンサを選択する．

5）センシングミドルウェアは，通知条件を付してセンシング動作の指示をセンサドライバに対して発行する．

6）センシングミドルウェアから指示を受けたセンサドライバはその指示に従ってセンシング処理を行い，その結果として発生したセンサイベントを,ミドルウェアに通知する．

7）6）のセンサイベントが，センシングミドルウェアを介してアプリに通知される．

7）の通知において，複合センサドライバのように通知条件判定を任せられる場合には，ドライバから上がってきたイベントをそのままアプリに通知する．一方で，アプリから指定された条件判定処理のできないセンサからのイベントに対してはセンシングミドルウェアが通知条件判定を代行し，アプリに指定された通知条件を満たしたらアプリに通知する．これにより，通知条件判定をサブシステムに任せられないような環境においても，アプリから指定された通知条件を満たした時のみ通知することができる．

このようにセンシングミドルウェアが端末の構成に応じてセンシングの振り分けを行うことで，個々のセンサの制御をアプリから隠ぺいし，アプリ開発者が端末の構成を意識しなくても端末の構成に応じてセンシングすることができる．

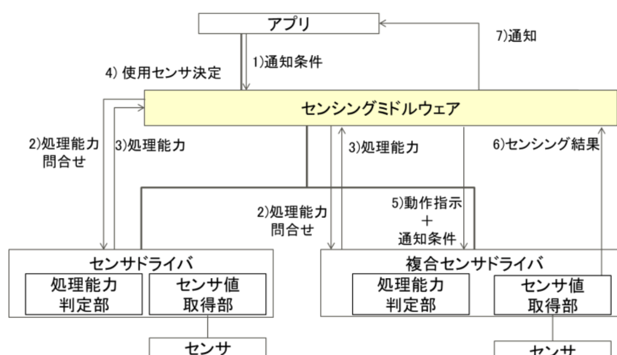

図 7　センシングミドルウェアの動作

5. 評価

提案した手法により，構成の異なる端末でも同一の API を用いてアプリ開発ができることと，通知条件判定を任せることのできるサブシステムを持つ端末で消費電力を抑えられることを示すため，提案した複合センシングサブシステムとセンシングミドルウェアを実装し，提案した API を用いたアプリを使って，電力の比較を行う．

5.1 想定シーン

複合センシングが用いられるシーンとして，人の行動や状況を継続的にとらえて建築物や設備の保守点検業務を支援するサービスを想定する．このサービスでは，以下のことができるようにする．

- 作業場所に到着したらその場所での作業リストを自動的に表示する
- 点検の結果問題がなければ，作業対象のタグをタッチするだけで正常であることを入力できるようにして入力作業を簡素化する

・作業時に点検箇所を飛ばしたらエラー通知を行い，作業の漏れを防ぐ

5.2 評価条件

想定したサービスでは，場所や物のセンシングが必要であるため，場所検知に Bluetooth Low Energy（BLE）ビーコン，作業対象物のタッチ検知に NFC タグを用いた実験を行った．ユーザーがアプリを起動すると，作業場所に行くように表示がされ，場所検知を行う．ユーザーが作業場所に到着すると，点検で使う道具を表示し，その道具にタッチすると点検内容として作業の順序が表示される．表示された順序の通り点検を行い，対象のタグをタッチした場合は作業終了時に正常終了したことが表示され，表示と異なるタグをタッチした場合はエラーを通知して正しい場所を点検するように指示される．

今回，スマート端末と，複合センシングサブシステムと，NFC タグを検知してタグ ID を無線で送信する NFC タグ検知デバイスを組み合わせて，想定シーンのセンシングをする実装を行った．スマート端末には，Android4.4.2 を搭載した Google Nexus5 (CPU Qualcomm Snapdragon 800)を用いた．また，複合センシングサブシステムとして，図 8 に示すデバイスを試作した．このセンシングサブシステムの諸元を表 1 に示す．センサとしては BLE モジュールと，WiFi モジュールと，加速度センサを搭載した．また，端末との通信用の BLE モジュールも搭載しており，端末 CPU の wake，通知条件や複合センシング結果の受け渡しを行う．コプロセッサとしては ARM 社 Cortex-M コアを搭載した STM 社のマイコンを用いた．このコプロセッサでは，BLE ビーコンのスキャン結果や NFC タグ検知デバイスが送信する無線電波のセンシング結果をもとに，場所検知とタッチを検知した結果が通知条件を満たしたか判定し、通知条件を満たした場合のみアプリに通知を行う．

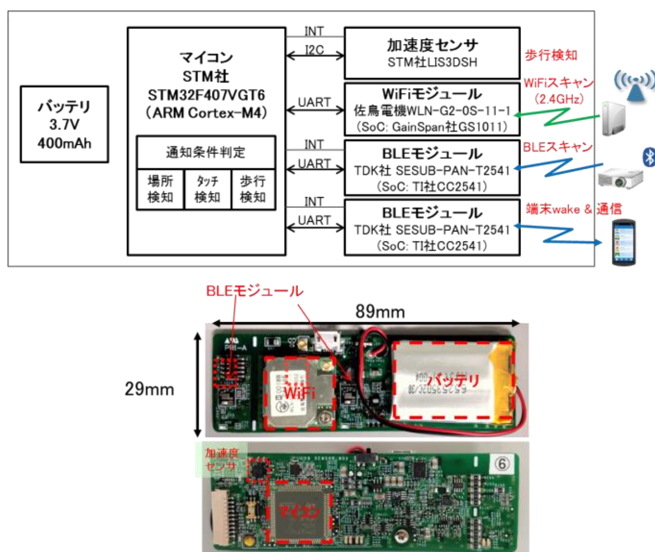

図 8　複合センシングサブシステム

表 1　センシングサブシステムの諸元

マイコン	STM 社 STM32F407VGT6 （ARM Cortex-M4 コア） 動作周波数は 30MHz に設定
BLE モジュール	TDK 社 SESUB-PAN-T2541 (SoC: TI 社 CC2541)
WiFi モジュール	佐鳥電機 WLN-G2-0S-11-1 (SoC: GainSpan 社 GS1011)
加速度センサ	STM 社 LIS3DSH
バッテリー	3.7V 400mAh

5.3 評価内容

構成の異なる 3 つ端末で，提案する API を用いた同一のアプリが動作することを確認する．また，通知条件判定をコプロセッサにさせることによる消費電力の削減効果を確認するため，表 2 に示す 3 つの実験で消費電流を測定する．

表 2　実験で用いる端末と各端末における処理の分担構成

	端末	処理の分担構成
実験 1	センシング用のコプロセッサを持たない端末	通知条件判定とセンサデータの処理の両方を端末で行う
実験 2	センサデータの処理だけができるコプロセッサを持つ端末	通知条件判定は端末で行い，センサデータの処理のみコプロセッサで行う
実験 3	複合センシングサブシステムを持つ端末	通知条件判定とセンサデータの処理の両方を複合センシングサブシステムで行う

実験 1 と 3 の比較により，通知条件判定を含めたセンシング処理のすべてをコプロセッサで行うことによる省電力効果を確認する．また，実験 1 と 2 の比較により，コプロセッサにセンサデータの処理をさせることによる省電力効果，実験 2 と 3 の比較により，通知条件判定をコプロセッサにさせることによる省電力効果を確認する．

なお，アプリとセンシングミドルウェアはすべての実験で同一のものを用いる．各実験の構成を図 9 に示す．

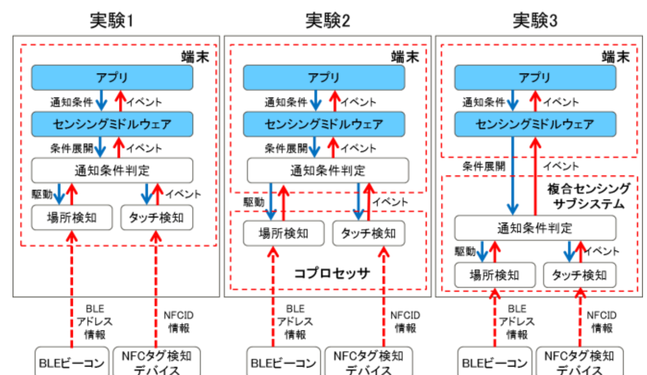

図 9　各実験における端末とコプロセッサや複合センシングサブシステムの処理範囲の構成

消費電流は，5.1 の想定シーンついて，タッチの検知を行っている時，場所の検知を行っている時，ディスプレイが ON の状態の画面確認時を測定する．通知条件判定をコプロセッサで行うことによる省電力効果を確認するために，タッチの検知を行っている時については，タッチを検知したイベントが発生して通知条件判定処理を行った時とそれ以外の時の両方の消費電流を測定する．

5.4 評価結果

実験 1～3 のすべての実験で，提案する API を用いてセンシングをする同一の業務支援アプリが正常に動作することを確認した．

また, 実験 1～3 における各場面のシステム全体の消費電流の測定結果と，5.1 の想定シーンでの 1 日の業務における平均消費電流を試算した結果を表 3 に示す．実験 1 では端末のバッテリー端で測定した消費電流，実験 2, 3 では端末のバッテリー端で測定した消費電流とセンシングサブシステムの消費電流の合計を示している．また，表 3 の想定シーンに占める時間は，平均消費電流を試算するにあたって想定した各場面の占める時間である．

表 3 消費電流の測定結果と試算結果

		実験 1	実験 2	実験 3	想定シーンに占める時間
タッチ検知	イベント発生時	58.8mA	55.6mA	19.5mA	0.24h
	上記以外	58.8mA	19.2mA	19.5mA	3.36h
場所検知		58.8mA	19.2mA	19.5mA	4h
画面確認		286.5mA	299.4mA	285.8mA	0.4h
平均消費電流(試算)		70.2mA	34.3mA	32.9mA	―

5.5 考察

実験 1 と 2 を比べると，タッチ検知のタッチイベント発生時以外と場所検知の消費電流が 58.8mA から 19.2mA へと減っている. この主な要因は, 約 16mA が CPU（Snapdragon 800）のスリープ分，約 20mA が BLE スキャンの時間の差によるものである．実験 1 では，CPU はセンサデータの処理を短い間隔でし続けるためスリープできないが，実験 2, 3 では, センサデータの処理をマイコン（STM32F407VGT6）で行うため， CPU はその間スリープでき，この差が約 16mA となった．また，BLE スキャンについては，実験 1 では常にスキャンを行うように制御をしたが，実験 2，3 では 1 秒間のうち 270ms だけスキャンをし，このことによる消費電流の差が約 20mA であった．本実験で用いたスマート端末では BLE スキャンを短い間隔で制御できなかったためこのような制御を行った．

実験 2 と 3 を比べると，タッチイベント発生時の消費電流が 55.6mA から 19.5mA へと減っている. この主な要因は約 16mA が実験 1, 2 の比較と同様の CPU のスリープ分で，約 20mA が CPU を wake する時のオーバーヘッドである．実験 2 ではタッチイベント発生時に通知条件判定をするために CPU が wake するが，CPU の wake 時はレジューム処理のため消費電力が大きくなる．この CPU を wake する時のオーバーヘッドによる消費電流の差が約 20mA となった．

1 日の保守点検業務での試算では，センシング処理をすべて端末でする実験 1 と比べ，複合センシングサブシステムを利用した実験 3 では 53.1％の消費電力を削減することができた．ただし，通知条件判定を端末で行った実験 2 とコプロセッサで行った実験 3 の差は 4.1％で，従来法であるコプロセッサにセンサデータの処理をさせたことによる効果が大半を占めている．これは，今回実験対象とした建築物や設備の保守点検業務では，従来法による効果が発揮される場面が多く，今回の提案方式が効果を発揮した場面は 8 時間中 0.24 時間であったためである. 本方式の効果は通知条件判定処理の占める時間に左右されるため，今後様々な利用シーンで検証していく．

6. まとめ

複数のセンシング結果の組合せ判定を CPU の代わりに CPU より省電力なコプロセッサにさせることで CPU が動作する頻度を減らし省電力化するサブシステムと，アプリ開発者が端末の構成を意識しなくても端末の構成に応じて適切に省電力化する API およびミドルウェアを提案した．

構成の異なる 3 種類の端末で同一の API を用いたアプリが動作することを確認した．また，消費電力の測定結果に基づく試算をし，通知条件判定処理をコプロセッサにさせることで消費電力が 4.1％削減されることを確認した．

今後は，今回の実験で想定した建築物や設備の保守点検業務以外の利用シーンについても，複合センシングをコプロセッサに任せることによる省電力効果を検証していく．

参考文献

1) Google Now
http://www.google.com/landing/now/
2) My Tracks
https://play.google.com/store/apps/details?hl=ja&id=com.google.android.maps.mytracks
3) More M7 details: data storage, battery life, Android motion coprocessor adoption
http://9to5mac.com/2013/09/29/more-m7-details-data-storage-battery-life-android-motion-coprocessor-adoption/
4) Apple iPhone 5s
https://www.apple.com/jp/iphone-5s/features/
5) Motorola X8 Mobile Computing System
https://www.motorola.com/us/X8-Mobile-Computing-System/x8-mobile-computing-system.html
6) Android SensorManager API
http://developer.android.com/reference/android/hardware/SensorManager.html
7) iPhone CoreMotion Framework
https://developer.apple.com/library/ios/documentation/CoreMotion/Reference/CoreMotion_Reference/_index.html