卒業論文 2004 年度 (平成 16 年度)

センサノード用オペレーティングシステムの設計と実装

指導教員

慶應義塾大学環境情報学部

徳田　英幸

村井　純

楠本　博之

中村　修

南　　政樹

慶應義塾大学 環境情報学部

須之内 雄司

suno@sfc.wide.ad.jp

卒業論文要旨　　2004年度(平成16年度)

センサノード用オペレーティングシステムの設計と実装

本論文では，センサノード用の軽量なプリエンプティブマルチスレッドオペレーティングシステムを提案する．開発者は本システムを利用することで容易にアプリケーションを開発可能になる．

ユビキタスコンピューティング環境では，ユーザや周囲の状況に即したサービス提供が行われる．サービス実現のために，実世界の情報を取得する機能を持つセンサノードが遍在し，ネットワークを介することで実世界の情報を細かに取得する．

現在多くのセンサノード用アプリケーションコードは，マイクロスレッドを採用した軽量なオペレーティングシステムか単純なライブラリを利用して書かれている．このような手法での開発には，割り込みやスケジューリングなどのハードウェアに関する知識が必要であり，センサノード用アプリケーション開発を未経験の開発者には難しい．多くの開発者はPCのオペレーティングシステムが提供するプリエンプティブマルチスレッドの利用経験がある．しかし，センサノードは小型化と省電力化のためにCPUやメモリなどのリソースが限られており，プリエンプティブマルチスレッド機構を搭載するにはコストが大きい．

本研究ではアプリケーションに特化した最適化により，コンテキストスイッチの処理をコンパイル時に削減可能なプリエンプティブマルチスレッドオペレーティングシステムを提案する．本機構を用いることで，マルチスレッド化によるメモリの使用と処理時間を削減する．プリエンプティブマルチスレッドを利用することで，開発者は容易にセンサノード用アプリケーションが開発可能となる．

本論文では，まずセンサノードのプログラミングモデルについての概要を整理する．次にセンサノード用プリエンプティブマルチスレッドオペレーティングシステムSNOS(Sensor Node Operating System)とアプリケーションとコンテキストスイッチ最適化ツールSNAO(Sensor Node Operating System)について述べる．その後SNOSとSNAOの設計と実装について説明する．最後に本システムを評価し，全体をまとめる．

慶應大学　環境情報学部
須之内 雄司

Abstract of Bachelor's Thesis

Design and Implementation of Multi-threaded Operating System for Sensor Nodes

This thesis presents a light-weight preemptive multi-thread operating system for sensor nodes. By using this system, developers can easily develop applications for sensor nodes.

In ubiquitous computing environment, services that adapts to state and location of people and objects within the environment will be provided. In order to provide such services, sensor nodes will be pervasively available, and the information of the environment can be collected through network.

Many applications for sensor nodes are written using light-weight operating system using micro-thread or a simple library. In order to use these methods, developers need to be aware of interrupts and scheduling, which is difficult for developers who are unexperienced with developing applications for sensor nodes.

Many developers experienced with preemptive multi-thread, which is provided by most operating systems running on PC's. But, sensor nodes have limited amount of CPU power and memory to reduce size and energy consumption, and preemptive multi-thread is too costly to be implemented on sensor nodes. The proposed system performs an application specific optimization at compile time to optimize speed and memory usage of contex switch.

In this thesis, first, clarifies constraints of programming models for sensor nodes. It then presents SNOS (Sensor Node Operating System), a preemptive multi-thread operating system, and SNAO (Sensor Node Application Optimizer), a tool for optimizing context switch. Next, it describes the design and implementation of SNOS and SNAO. Finally, it shows evaluation of the system and concludes.

Yuji Sunouchi
Faculty of Environmental Information Keio University

目 次

図 目 次

表 目 次

第1章　序論

1.1 本研究の背景

ユビキタスコンピューティング環境では，ユーザや周囲の状況に即したサービス提供が行われる．例として，人が向いている方向に TV の画面を映し出すアプリケーションや，忘れ物がどこにあるかを調べてくれるアプリケーションなどの例が考えられる．これらのサービス実現のため，位置情報取得機能，加速度取得機能，無線通信機能などを持つセンサノードが遍在し，ネットワークを介することで実世界の情報を細かに取得する．

ここ数年で Mica2 Mote[1] や Smart-Its[3] などの小型センサノードが研究用に商品化されてきた．これらはセンシング機能や通信機能の他に，PC などに比べて低性能なプロセッサや低容量の RAM を搭載している．そのため，アプリケーション開発者はアプリケーションを ROM に書き込み，自由に制御できる．

1.2 問題意識

現在，多くのセンサノード用アプリケーションコードは，マイクロスレッドを採用した軽量なオペレーティングシステムか，単純なライブラリを利用して書かれている．このような手法での開発には，割り込みやスケジューリングなどのハードウェアに関する知識が必要なため，センサノード用アプリケーション開発を未経験の開発者には難しい．

多くの開発者は PC のオペレーティングシステムが提供するプリエンプティブマルチスレッドを利用してアプリケーションを開発している．しかし，センサノードは小型化と省電力化のために CPU やメモリなどのリソースが限られており，プリエンプティブマルチスレッド機構を搭載するにはメモリや処理時間の消費が大きい．そこで，センサノード用に軽量化したプリエンプティブマルチスレッドオペレーティングシステムを利用することで開発者はより効率よくセンサノード用アプリケーションを開発できる．

1.3 目的と意義

本研究の目的はアプリケーション開発者がセンサノード用アプリケーションを容易に開発できるオペレーティングシステムの構築である．本研究ではセンサノード向けオペレーティングシステム SNOS(Sensor Node Operating System) を提供する．本システムは Unix に似たマルチスレッドインターフェースを提供し，センサノード用アプリケーション開発が未経験の開発者でも容易にアプリケーション開発を行うことができる．

また，本研究は SNOS 用に作成されたアプリケーションを最適化するツール SNAO(Sensor Node Application Optimizer) を提供する．SNAO を用いてアプリケーションを最適化することで，マルチスレッド化によるメモリの使用や処理時間を大幅に削減する．

1.4 本論文の構成

本論文では，第 2 章において，センサノードのハードウェアとアプリケーションについて詳しく説明し，センサノード用オペレーティングシステムとその先行研究について述べる．第 3 章ではまず，本研究の目的と機能要件を整理し，次に本研究の用いるアプローチについて述べる．第 4 章でセンサノード用プリエンプティブマルチスレッドオペレーティングシステム SNOS とセンサノード用アプリケーション最適化ツール SNAO の設計について述べ，第 5 章で実装について述べる．そして，第 6 章で SNOS と SNAO を評価し，第 7 章で本論文をまとめる．

第2章　センサノードのプログラミングモデル

本章ではセンサノードのプログラミングモデルについて述べる．まず，ハードウェアによる制約について述べ，次にソフトウェアによる制約について述べる．さらに各プログラミングモデルとそれぞれのプログラミングモデルを採用した先行システムについて述べる．

2.1 ハードウェアによる制約

本節ではセンサノードのハードウェアによる，プログラミングモデルの制約について述べる．まず，センサノードの基本性能による制約について述べ，次にセンサノードの I/O デバイスによる制約について述べる．

2.1.1 基本性能による制約

既存のセンサノードのハードウェアとしては Mica2 Mote，Smart-Its，U^3[7] などがある．表 2.1 のように，センサノードは小型化と長時間電池で駆動させるため，ハードウェアの性能は PC と比べて低い．

将来的にはムーアの法則によりセンサノードの性能が向上し，アプリケーションに対する制約がゆるくなると予想される．しかし，スマートダスト [6] のようなサイズを目指して同一性能，又はより低い性能で小型化，省電力化されていくことも予想されるため，現在の制約で動作することが今後も必要である．

表 2.1: 既存のセンサノードの基本性能

センサノード	CPU	速度	メモリ	コード領域
Mica2 Mote	Atmega128L	4MHz(PIC 16MHz 相応)	4KB	128KB
Smart-Its Particle 2/29	PIC18F6720	25MHz	4KB	128KB
U^3	PIC18F452	20MHz	1.5KB	32KB

2.1.2 I/O デバイスによる制約

センサノードは温度・湿度・光度など各種センシングデバイスをいくつか持つ。また、センサから取得したデータを PC などに送るために無線通信機能やシリアル通信機能などを有する．

PC の I/O デバイスと違い，センサノードの I/O デバイスにはコントローラが用意されてないものがある．コントローラがない I/O デバイスの制御は CPU で動作するアプリケーションが行う．よって頻繁に処理を必要とする I/O デバイスを持つセンサノードではアプリケーションが頻繁に I/O を処理しなくてはならない．例として 19200bps で通信する無線通信機能を持つセンサノードは約 50 マイクロ秒ごとに 1bit のデータを処理する必要がある．

2.2　ソフトウェアによる制約

本節ではまずセンサノードのソフトウェアによるプログラミングモデルの制約について述べる．まずオペレーティングシステムによる制約について述べ，次にアプリケーションによる制約について述べる．

2.2.1　オペレーティングシステムによる制約

オペレーティングシステムはスケジューラやデバイス操作インターフェースを提供する．プログラミングモデルはスケジューラによって決定される．アプリケーションがスケジューラを実装することで別のプログラミングモデルを利用することは可能だが，リソースを余分に消費するため，汎用的なモデルが望ましい．

2.2.2　アプリケーションによる制約

センサノード用アプリケーションは環境の情報を集めることを主とする．例として位置を測定するアプリケーションや定期的に温度を計るアプリケーションなどが挙げられる．これらのアプリケーションは取得したデータを無線やシリアルを通じて PC などに送る．

図 2.1 のように複数のセンシングデバイスを持つ複合センサでは温度を定期的に送る機能と湿度を定期的に送る機能など独立した複数の機能を持つアプリケーションが実行されることある．また，センシングデバイスを一つしか持たないセンサノードに関しても，マルチホップ通信を行うアプリケーションではセンサデータを送信をする機能と他ノードから届いたデータをブリッジする機能が共存する．このようにアプリケーションを作成する際には機能の独立性が重要となる．

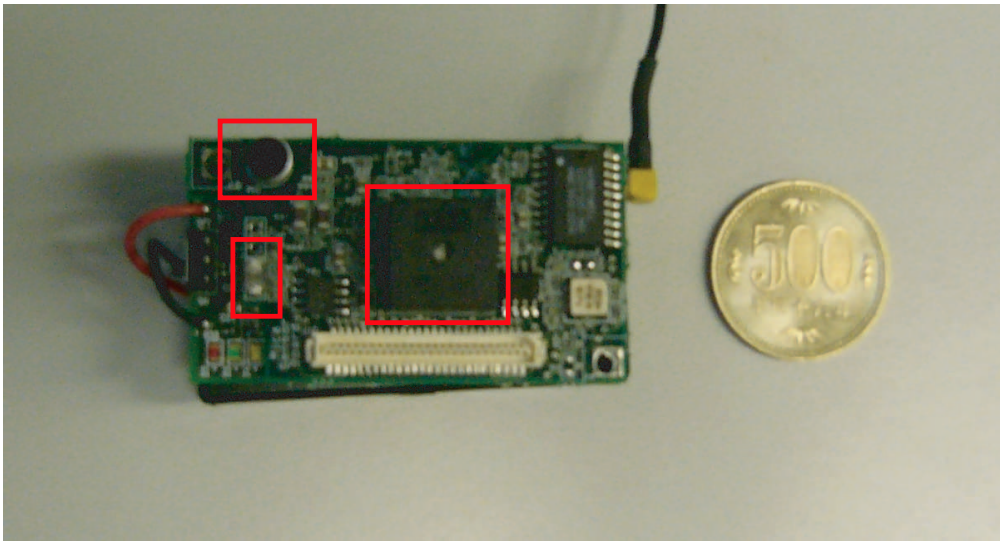

図 2.1: 複合センサノード

2.3 センサノードのプログラミングモデル

プログラミングモデルにはポーリング，割り込み，マイクロスレッド，プリエンプティブマルチスレッドなどがある．以下にそれぞれの特徴と各プログラミングモデルを採用しているオペレーティングシステムについて述べる．

2.3.1 ポーリングと割り込み

オペレーティングシステムがスケジューラを提供しない場合，アプリケーションはポーリングと割り込みを用いてプログラムを書く．ポーリングするプログラムではI/Oが入出力可能であるかをループで監視し，入出力可能な時点で入出力を行う．図2.2にその様子を示す．割り込みはデバイスのイベントにより発生し，その時実行中のプログラムを一時停止し，割り込みハンドラの処理を実行し，割り込みハンドラの処理が終了次第，以前実行していたプログラムの処理を続ける．

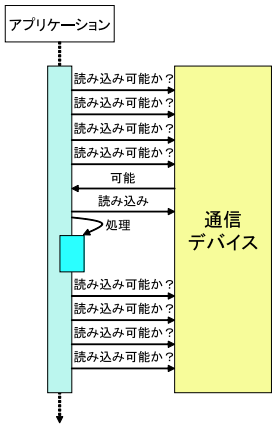

図 2.2: ポーリング

ポーリングと割り込みはCPUの基本機能であるため，リソースの消費は比較的小さくなる．しかし機能拡張の手段を自ら構築しなくてはいけないため機能の独立性は低い．ポーリングはPC用アプリケーションでも多く利用されているため，わかりやすい．しかし割り込みは図2.3のように，割り込みハンドラの処理中に割り込みを禁止して，長い処理やループを行うと他の割り込みが受けられなくなる．逆に割り込みハンドラの処理中に割り込みを許可すると，再帰的に割り込みが発生し，メモリスタックがあふれることがある．アプリケーション開発者は割り込みの性質を理解しなくてはならない．

ポーリングを採用した例としてSmart-Itsのソフトウェアプラットフォームを挙げる．Smart-ItsはLancaster大学，ETH Zurich，Karlsruhe大学，Interactive Institute，VTT Electronicsが開発したセンサノードである．Smart-Itsのソフトウェアプラットフォームはセンサノードの各デバイスに対する操作関数を提供するがスケジューラを持たず，アプリケーション開発者はポーリングを用いてアプリケーションを書く．

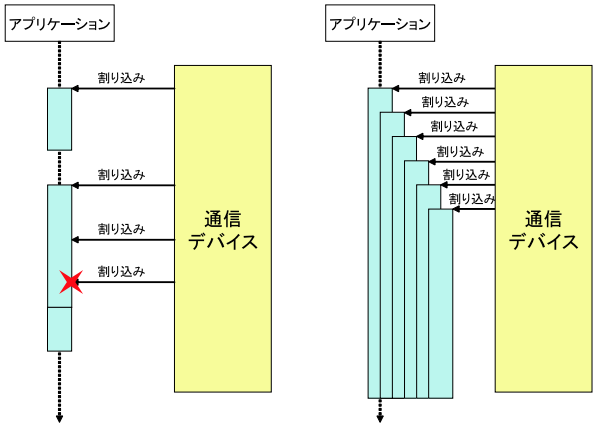

図 2.3: 割り込み

2.3.2 マイクロスレッド

マイクロスレッドのモデルでは，タスクとイベントとキューの 3 要素で構成される．プログラムのメインスレッドがキューに入っているタスクを順次処理していく．タスクはイベントや他のタスクにより追加される．イベントとはハードウェア割り込みなどにより発生する処理で，メインスレッドに割り込める．イベントは割り込みを長時間禁止しないように即座に制御を返す必要がある．そのため，イベントでは時間のかかる処理は行わず，代わりに処理用のタスクをキューに追加する．イベントとタスクの構成を図 2.4 に示す．

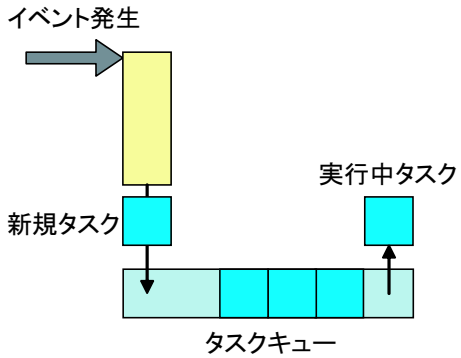

図 2.4: イベントとタスク

マイクロスレッドは個々のタスク開始時にメモリスタックが伸び，タスク終了時にメモリスタックが縮むためリソースの消費は大きくない．また，機能を拡張するにはタスクとイベントを新規に追加するだけなので拡張性が高い．しかしマイクロスレッドではキューに加えられたタスクを FIFO で処理していくため，タスクが内でループ処理を行うと，その間他のタスクは実行されず，イベント他タスクがキューに追加され，キューが一杯になってしまう可能性がある．図 2.5 にその様子を示す．この問題を解決するにはループをタスクに分割し，図 2.6 のように，分割されたタスクが終了する際に次のループ用にタス

クを再度追加するようなコードを書かなくてはならない．このように，マイクロスレッドを利用するアプリケーションを開発する際には，マイクロスレッドの性質を理解していなくてはならない．

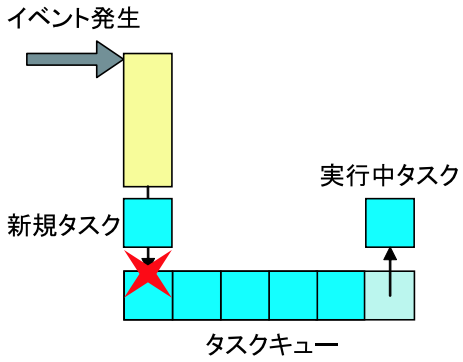

図 2.5: キューがあふれる

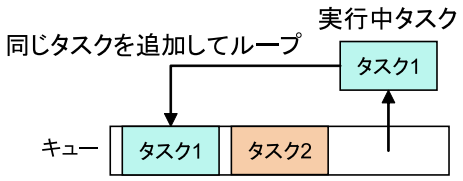

図 2.6: 分割したタスクによるループ

マイクロスレッドを採用した例として TinyOS[5] を挙げる．TinyOS は UC Berkley で開発されたワイヤレスセンサネットワーク向けの OS である．TinyOS のアプリケーションは NES-C[4] で開発する．NES-C は C 言語に似た独自のオブジェクト指向言語で，モジュールと呼ばれるタスクとイベント処理の集合を記述し，組み合わせることでアプリケーションを構築する．

2.3.3 プリエンプティブマルチスレッド

プリエンプティブマルチスレッドは一般的な PC のオペレーティングシステムで用いられているモデルである．スレッドはプログラムの実行単位であり，図 2.7 のように短い間隔でコンテキストスイッチを行うことで，並列動作させる．アプリケーションはスレッドの切り替えタイミングを明示的に指定する必要がなく，OS によって自動的にスレッドが切り替えられる．また，全てのスレッドがブロックした際に CPU をスリープさせ，イベントを待つことで電力消費を抑えられる．

コンテキストスイッチの際に各スレッド固有の情報であるレジスタなどの値をメモリに退避しなくてはならないため，処理のオーバヘッドが生じ，生成するスレッド数と比例してメモリ領域が必要となる．よってリソースの消費は大きい．しかし機能の拡張は新しい

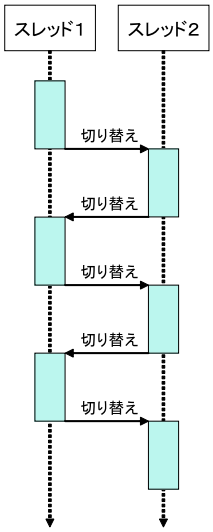

図 2.7: プリエンプティブマルチスレッド

スレッドを追加することで実現可能であるため，独立性が高い．また，多くの開発者が利用するプログラミングモデルである．

プリエンプティブマルチスレッドを採用した例としてMantis[2] を挙げる．Mantis はコロラド大学が開発しているセンサノード用オペレーティングシステムである。アプリケーションのプロトタイプ作成に主観を置き，Unix と似た API を提供している．Mantis は Mica2 Mote に対応しており，エミュレータが Linux 上で動作する．

2.4 本章のまとめ

本章ではまず，センサノードのハードウェアについて述べ、センサノードのオペレーティングシステムについて述べた。さらにプログラミングモデルとそれぞれのモデルを用いた先行システムの例を挙げ，センサノードのアプリケーションについて述べた．表 2.2 にプログラミングモデルの比較をまとめる．

次章では，本研究の構成について詳しく述べる．

表 2.2: プログラミングモデルの比較

モデル	省リソース	機能の独立性	PC での利用
ポーリング・割り込み	○	×	△
マイクロスレッド	△	○	×
プリエンプティブマルチスレッド	×	○	○

第3章　センサノード用オペレーティングシステム

前章ではセンサノードのプログラミングモデルについて述べた．本章ではまず，本研究の目的を延べ，目的を実現するための機能要件を整理する．次に本研究のアプローチについて述べる．

3.1 採択するプログラミングモデル

現在，多くのセンサノード用アプリケーションはセンサノードの少ないリソースで動作するよう，ポーリングと割り込みやマイクロスレッドを用いている．しかしこれらの手法は 2.3.1 と 2.3.2 で述べたように開発者がそれぞれのモデルの特性を理解しなくてはならない．

プリエンプティブマルチスレッドは PC 用アプリケーションで多く利用されているため，センサノードのアプリケーション開発が未経験の開発者でも理解できる．しかし 2.3.3 で述べたように，プリエンプティブマルチスレッドはコンテキストスイッチによるリソースの消費が大きく，センサノードでの動作に適していない．

本研究ではアプリケーション開発者が容易に開発を行うことを目的とし，多くの開発者が利用経験のあるプリエンプティブマルチスレッドを提供する軽量なセンサノード用オペレーティングシステムを構築する．

3.2 機能要件

本節ではセンサノード用プリエンプティブマルチスレッドオペレーティングシステムの機能要件を整理する．プリエンプティブマルチスレッドは他のモデルに比べ処理時間やメモリなどのリソースを多く必要とする．特にリソースが必要となるのはコンテキストスイッチである．また，割り込みなどに親しみがない開発者でも容易に開発できるよう，割り込みハンドラを書かずにアプリケーション開発ができる機能が必要である．機能要件として，短い間隔のデータの対応，コンテキストの退避メモリの削減，コンテキストスイッチの処理オーバヘッドの削減が挙げられる．以下でそれぞれの要件について述べる．

短い間隔のデータへの対応

2.1 で述べたように 19200bps で通信する無線モジュールを持つセンサノードでは，アプリケーションが約 50 マイクロ秒ごとにデータの処理を行わなくてはならない．時間のかかる処理中でも頻繁に到来するデータを処理しなくてはならない．

コンテキストの退避メモリ削減

スレッドはそれぞれ実行コンテキストを持つ．実行コンテキストには実行中のコードのアドレス，スタックレジスタ，汎用レジスタなどが挙げられる．実行していないスレッドのコンテキストは後に実行する際に復元できるようメモリに退避されている．センサノードの限られたメモリで動作するよう，退避する情報を最低限におさえなくてはならない．

コンテキストスイッチの処理オーバヘッドの削減

コンテキストスイッチでは，実行中のスレッドのコンテキストを退避し，次に実行するスレッドのコンテキストを復元する．コンテキストスイッチは定期的に発生し，アプリケーション本体が実行できる CPU 時間を奪う．アプリケーションが実行できる時間をより長くするためにコンテキストスイッチ処理を短くしなくてはならない．

3.3 アプローチ

本研究では 3.2 で述べた機能要件を満たすオペレーティングシステム SNOS(Sensor Node Operating System) とアプリケーション最適化ツール SNAO(Sensor Node Application Optimizer) を構築する．SNOS はプリエンプティブマルチスレッドとデバイスドライバを提供し，SNAO はアプリケーションを解析し，アプリケーションとコンテキストスイッチ処理を最適化する．

SNOS は C コンパイラでリンク可能なライブラリとして提供され，SNAO はアプリケーションのアセンブラコードを解析するパーサの形で提供される．本システムは開発者のコードをアセンブラに変換し，SNAO により最適化を行う．その最適化結果を元に SNOS を最適化し，最適化された開発者のコードと SNOS をリンクすることでアプリケーションを作成する．

3.4 デバイスドライバ

スレッドがポーリングで 50 マイクロ秒間隔で発生するデバイスの I/O 処理を行うには，50 マイクロ秒以内に I/O を処理し，コンテキストスイッチを行い，別の処理を行い，再度コンテキストスイッチを行わなくてはならないため，センサノードで用いられるような CPU では現実的ではない．デバイスドライバはデバイスを監視するスレッドが実行していない間も非同期に I/O 処理を行う．デバイスドライバはトップハーフとボトムハーフで構成される．

トップハーフ

トップハーフはアプリケーションからの呼び出しにより実行される処理である．トップハーフはデバイスに対して入出力を行い，その結果をアプリケーションに返す．USART や無線の受信などのデータを待つ処理は，トップハーフから直接デバイスを操作せず，ボトムハーフがデータをメモリにバッファするまで待つ．

ボトムハーフ

ボトムハーフはハードウェア割り込みに対する割り込みハンドラとして実装される．例として，前述したように USART や無線のデータ到着などでデータをバッファする機能などが挙げられる．LED のように，ピンのオンとオフを変更するだけのデバイスドライバはボトムハーフを持たない．

3.5 割り込み可能なコンテキストスイッチ

コンテキストスイッチは時間のかかる処理なため，割り込みが禁止されているとデバイスドライバのボトムハーフが短い間隔で発生する割り込みを処理しきれなくなる．コンテキストスイッチ中に割り込みを受けるために，SNOS ではコンテキストスイッチ中の割り

込みを許可することでボトムハーフの処理を優先的に実行可能にする．アプリケーションとデバイスドライバとコンテキストスイッチ処理の割り込み関係を 3.1 にまとめる．

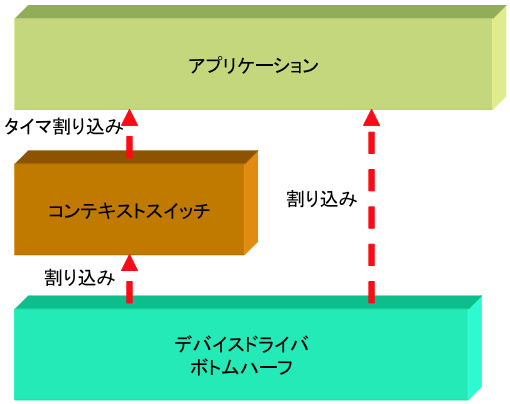

図 3.1: 割り込みの許可

3.6　コンテキストスイッチの高速化

SNAO アプリケーションのアセンブラコードを解析し，最適化することで SNOS のコンテキストスイッチ処理を高速化する．本節ではその手法について説明する．

3.6.1　コンテキストスイッチ処理の削減

コンテキストスイッチの処理は主にコンテキストの退避，次に実行するスレッドの選択，コンテキストの復元の三つのステップで構成される．ラウンドロビンなどの単純なスケジューリングアルゴリズムでは，次に実行するスレッドの選択は処理時間は短く，コンテキストの退避と復元がコンテキストスイッチ処理にかかる時間の多くを占める．

退避・復元するコンテキストにはプログラムカウンタ，スタックレジスタ，汎用レジスタなどが挙げられる．汎用レジスタは CPU によって 32 本〜64 本あり，コンテキストの退避・復元の大半を占める．SNAO は退避・復元する汎用レジスタの数を減らすことでコンテキストスイッチを高速化する．

3.6.2　退避・復元しなくて良い汎用レジスタ

プリエンプティブマルチスレッドでは，いつスレッドがタイマ割り込みによって中断され，別のスレッドに実行権限が委譲されるかがわからない．よって，利用中のレジスタは別スレッドによって上書きされても復元できるように，コンテキストスイッチ時に退避・復元される．しかし，１スレッドのみで利用されているレジスタと全く利用されていない

レジスタは別スレッドによって上書きされることはない．このようなレジスタはコンテキストスイッチで退避・復元しなくて良い．

3.6.3 アプリケーションの最適化

汎用レジスタの多くは特別な用途が決まっていないため，利用するレジスタを変更しても同様に動作する．スレッドごとに使用するレジスタが重複しないようにレジスタを変更することで，退避・復元しなくてはならないレジスタを削減できる．C 言語のコードでは使用するレジスタを指定できないため，SNAO はアプリケーションのアセンブラコードを元にレジスタの変更を行う．

3.7 本章のまとめ

本研究の目的は軽量なプリエンプティブマルチスレッドオペレーティングシステムの構築である．本研究ではプリエンプティブマルチスレッドを提供するオペレーティングシステム SNOS とコンテキストスイッチ最適化ツール SNAO を構築する．

SNOS はコンパイラでリンク可能な形で提供され，割り込み可能なコンテキストスイッチは割り込み可能センサノード用オペレーティングシステム SNOS を提供する．また，SNAO により，アプリケーションとコンテキストスイッチを最適化する．

次章では，SNOS 及び SNAO の設計について述べる．

第4章　設計

前章では，プリエンプティブマルチスレッドオペレーティングシステム及びアプリケーション最適化手法について述べた．本章ではプリエンプティブマルチスレッドオペレーティングシステムSNOS及びアプリケーション最適化ツールSNAOについて詳細を述べる．まず，システム全体の概要について述べる．次にSNOSを構成するモジュールについて述べ，最後にSNAOの構造について述べる．

4.1 全体概要

本節ではまず，SNOS 及び SNAO の利用方法について述べる．

SNOS は C 言語のコードで提供され，アプリケーション開発者は SNOS のが提供するシステムコールを用いてアプリケーションを開発する．SNAO を用いて SNOS とアプリケーションのコードを最適化し，コンテキストスイッチを最適化する．SNOS と SNAO の利用は以下の手順で行われる．

アプリケーションのコードをアセンブリコードに変換
SNAO はアセンブリコードの最適化を行うため，アプリケーションのコードをコンパイラでアセンブラに変換する．

アセンブラコードの最適化
前の手順で生成したコードを SNAO で解析し，マルチスレッドに最適化されたアセンブラコードに変換し，オブジェクトファイルを生成する．また，最適化の結果を SNOS に反映するために最適化設定ファイルを生成する．

SNOS の最適化
前の手順で生成された最適化設定ファイルを元にコンテキストスイッチを最適化し，オブジェクトファイルを生成する．

リンク
アプリケーションのオブジェクトコードと SNOS のオブジェクトコードをリンクし，アプリケーションバイナリを生成する．

4.2 SNOS の設計

SNOS は短い間隔で発生する割り込みに対応するためにデバイスドライバを提供し，コンテキストスイッチ中に割り込み可能である．割り込み許可のルールとして，SNOS には図 4.1 のようにアプリケーションモード，カーネルモード，コンテキストスイッチモード，割り込みモードの四つの実行モードがある．四つの実行モードの遷移例を図 4.2 に示す．以下に四つのモードの詳細について述べる．

4.2.1 アプリケーションモード

アプリケーションのコード全てはアプリケーションモードで実行される．アプリケーションモードの処理は優先度が最も低く，あらゆる割り込みによって割り込まれる．アプリケーションモードの処理はシステムコールの呼び出しによりカーネルモードに遷移し，タイマ割り込みによってコンテキストスイッチモードに遷移し，その他割り込みによって割り込みモードに遷移する．

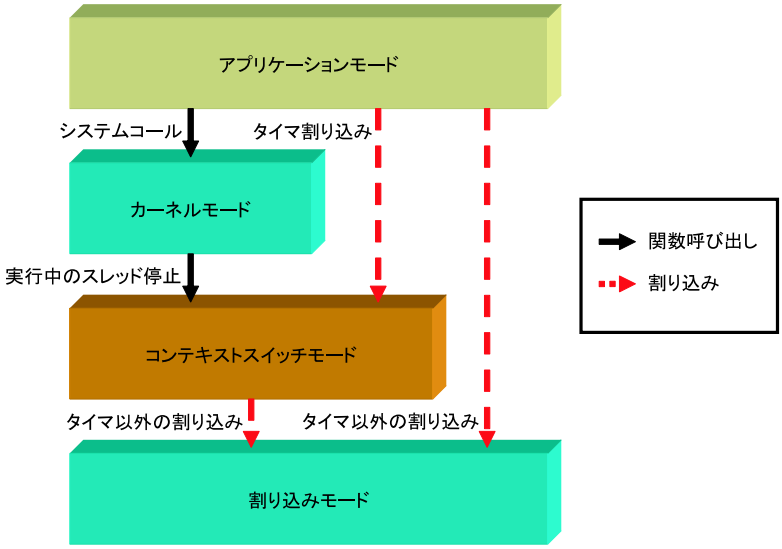

図 4.1: 実行モード

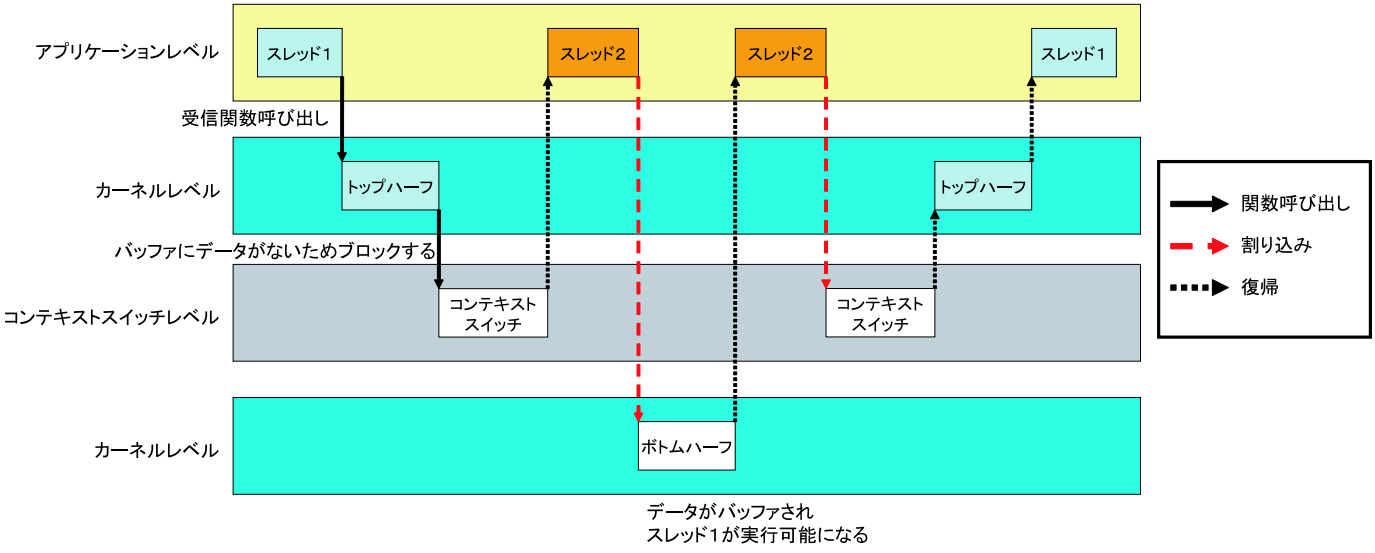

図 4.2: 実行モードの遷移例

4.2.2 カーネルモード

デバイスドライバのトップハーフなど，アプリケーションから呼び出されるシステムコールの処理はカーネルモードで実行される．カーネルモードの処理は割り込まれることはない．同期 I/O など，スレッドがブロックするカーネルモードの処理はコンテキストスイッチのためにコンテキストスイッチモードに遷移する．カーネルモードの処理が終了したスレッドはアプリケーションモードに遷移する．

4.2.3 コンテキストスイッチモード

コンテキストスイッチの処理はコンテキストスイッチモードで実行される．コンテキストスイッチはカーネルモードの処理から明示的な呼び出しとタイマ割り込みによって呼び出される．タイマ割り込みはアプリケーションモードの処理に割り込むことはできるが，カーネルモードの処理と割り込みモードの処理には割り込めない．コンテキストスイッチ終了後は再開するスレッドが元々動作していた実行モードに遷移する．

4.2.4 割り込みモード

デバイスドライバのボトムハーフは割り込みモードで実行される．ボトムハーフの処理はアプリケーションモードの処理とコンテキストスイッチモードの処理に割り込むことができるが，データの整合性のために，カーネルモードの処理には割り込めない．ボトムハーフの処理終了後は以前の実行モードに遷移する．

4.3 SNAO の設計

本節では SNAO の設計について述べる．まず SNAO の最適化手順について述べる．SNAO によるアプリケーション最適化は構文解析，関数の複製，呼び出し関数の変更，利用レジスタの変更，最適化設定ファイルの生成の五つの段階で構成される．次に SNAO の制約について述べる．

4.3.1 構文解析

アプリケーションのアセンブラコードをトークンに分解し，関数の開始と終了の範囲や関数呼び出しを行っている箇所を調べる．スレッドのエントリポイント関数から呼び出される関数を再帰的に追うことで各スレッドのコールグラフを生成する．

4.3.2　関数の複製

同じ関数が複数のスレッドから呼び出されると，その関数で使うレジスタが複数スレッドで利用されることになる．この段階ではスレッドごとの最適化が行えるように関数をスレッドごとに複製する．

各スレッドで呼ばれている関数をコールグラフを元に調べ，複数のスレッドが同じ関数を呼び出している場合，重複している関数を別の名前で複製する．また，複製した関数から呼び出される関数もコールグラフを追い，再帰的に複製を繰り返す．複数スレッドから呼び出される関数全てを複製後にスレッドのエントリポイントとなる関数から再帰的に関数呼び出しのラベルを置き換えていく．図 4.3 に例を示す．

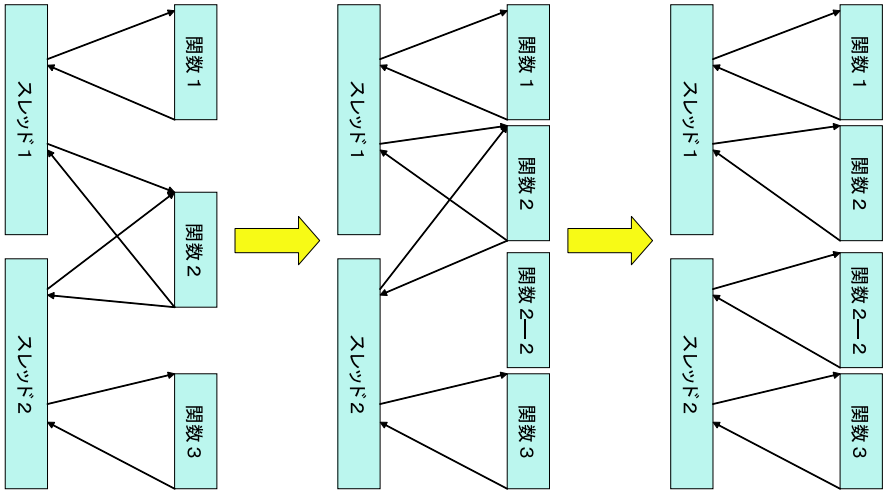

図 4.3: 関数の複製

4.3.3　利用レジスタの変更

複数のスレッドから利用されているレジスタがある場合，スレッドごとに利用するレジスタを変更する．関数間のレジスタの整合性を取るために関数ごとの最適化は行わず，スレッド単位で最適化を行う．

各スレッドで利用されているレジスタをスレッドのエントリポイントからコールグラフを追い，再帰的に調べ，各レジスタがいくつのスレッドから参照されているかを数える．2 つ以上のスレッドから参照されているレジスタは利用されていないレジスタに置き換える．図 4.4 に利用レジスタの変更例を示す．

4.3.4　最適化設定ファイルの生成

再度各スレッドで利用されているレジスタを調べ，スレッド間で利用が重複しなかったレジスタと全く利用されなかったレジスタを退避しないレジスタとしてまとめる．退避し

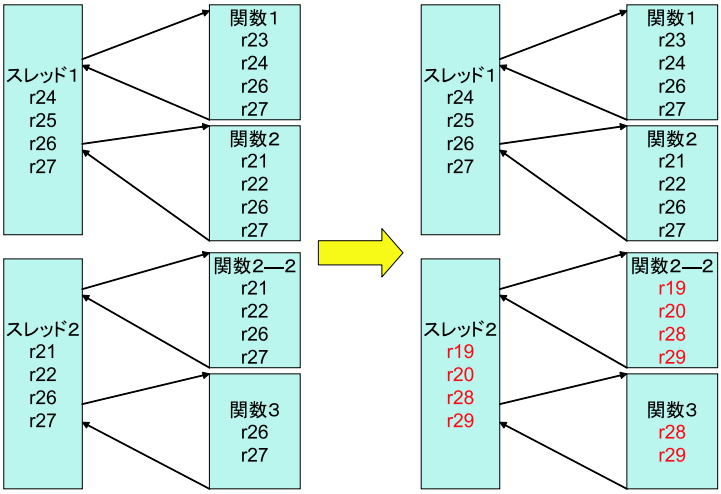

図 4.4: 利用レジスタの変更

なくて良いレジスタは SNOS から include されるヘッダファイルに保存され，SNOS のコンパイル時にスケジューラのスレッド切り替え関数に反映される．

4.3.5 SNAO の制約

SNAO で最適化を行うアプリケーションコードには以下の制約がある．

- 間接関数呼び出しの利用

  関数ポインタを通じた関数呼び出しはサポートしない．

- ライブラリファイルの利用

  SNAO はアセンブリコードの最適化を行うため，ライブラリファイルの最適化が行えない．

4.4 本章のまとめ

本章では SNOS と SNAO の利用方法について述べ，SNOS のモジュール構成と各モジュールの詳細，そして SNAO の最適化の詳細について述べた．次章では SNOS と SNAO の実装について述べる．

第5章　実装

前章ではSNOS,SNAOの設計及び仕組みについて述べた．本章ではSNOS及びSNAOの実装概要及び各モジュールの実装について詳しく述べる．

5.1 SNOS の実装

SNOS は Mica2 Mote 上で実装した．表 5.1 に Mica2 Mote 及びアプリケーション作成に用いたコンパイラの詳細を示す．

表 5.1: SNOS の実装環境

項　　目	環　　境
CPU	Atmega128L
Clock	4MHz
メモリ	4KB
コード領域	128KB
コンパイラ	win-avr(avr-gcc) 3.3.2

SNOS は図 5.1 に示すような 4 つのモジュールから構成される．アプリケーションはスレッドモジュール，セマフォスモジュール，I/O モジュールを通じて本ライブラリを利用する．

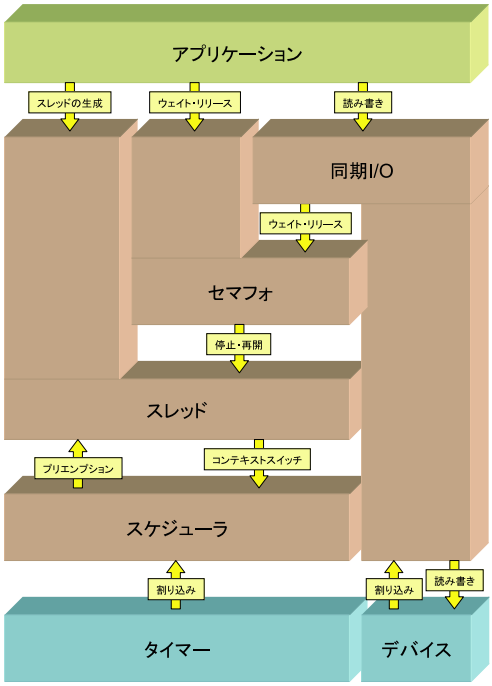

図 5.1: モジュールの構成

5.1.1 スレッド

スレッドはプログラムの実行単位であり，それぞれのスレッドが独自のレジスタの値とスタックを持つ．実行するスレッドはセマフォや同期 I/O によって切り替えられる他，一定時間ごとにスケジューラによって切り替えられる．スレッドモジュールはスレッド固有の情報を保存するためのスレッド情報構造体を保持し，スレッド生成関数，スレッド停止関数，スレッド再開関数を提供する．

スレッド情報構造体

スレッド情報構造体は各スレッドの実行情報を保持する．スタックレジスタの値を保存するための領域の他，実行可能かのフラグやブロックの原因となったセマフォの ID と待機番号を保持する．

スレッド生成関数

スレッド生成関数はアプリケーションからスレッドモジュールにアクセスできる唯一のインターフェースである．スレッド生成関数の引数にはエントリポイントとなる関数のアドレス，そしてスレッドに割り当てるスタックのアドレスとそのサイズを渡す．スレッド生成関数のプロトタイプ宣言を図 5.2 に示す．新しく作成するスレッドのスタック領域をアプリケーションが指定することで，最低限のメモリ領域を割り当てることができる．本ライブラリはメモリサイズの節約のため，スレッドの数をコンパイル時に決定し，実行時はそれ以上のスレッドは作成できない．

スレッド生成関数はスレッド情報構造体を初期化し，実行中スレッドのリストに加える．実行中スレッドのリストはスレッド情報構造体の循環リストで管理される．

```
/* スレッド生成関数 */
char createThread(
  /* エントリポイント */
  void (*entrypoint)(char),
  /* スレッドのスタック領域 */
  char* stack,
  /* スタック領域のサイズ */
  char stacksize);
```

図 5.2: スレッド生成関数

スレッド停止関数

スレッド停止関数は実行中のスレッドを停止する．この関数はセマフォモジュールを通じて呼び出される．この関数は引数として停止の原因となったセマフォの ID と待

機番号をとり，実行中のスレッドのスレッド情報構造体に保存し，実行可能状態から停止状態に変更し，スケジューラモジュールのスレッド切り替え関数を呼び出す．

スレッド再開関数

スレッド再開関数は引数で指定されたセマフォの ID と待機番号のスレッドを停止状態から実行可能状態に変更する．実行可能状態になったスレッドは，後にスレッド切り替えにより実行される．

5.1.2 スケジューラ

スケジューラモジュールはスレッドの管理を行う．スケジューラはスレッド切り替え関数を提供する．スレッド切り替え関数はスレッドモジュールのスレッド停止関数とタイマの割り込みで呼び出される．スレッド切り替え関数はスレッド固有情報の保存，実行スレッドの選択，スレッド固有情報のロードの 3 段階で構成される．

スレッド固有情報の保存

最初に，スレッド切り替え関数は実行していたスレッド固有の情報を保存する．スレッド固有の情報には汎用レジスタ，スタックレジスタ，ステータスレジスタ，プログラムカウンタがある．スレッド切り替え関数は汎用レジスタとステータスレジスタをスタックに push し，スタックレジスタをスレッド情報構造体に保存し．プログラムカウンタはスレッド切り替えの関数の呼び出しスタック上に push される．

実行スレッドの選択

次に，スレッド切り替え関数は実行するスレッドを選択する．センサノードの限られたメモリサイズとプログラムサイズで動作させるため，スケジューリングアルゴリズムはシンプルなラウンドロビンを採用した．スレッド切り替え関数はスレッド情報構造体の循環リストを辿り，実行可能状態のスレッドを探す．全てのスレッドが停止状態の場合，アイドルし，割り込みによってスレッドが再開されるのを待つ．

スレッド固有情報の保存

最後に，スレッド切り替え関数は次に実行するスレッドの固有情報を CPU にロードする．スケジューラはスレッド情報構造体からスタックポインタを CPU にロードし，汎用レジスタとステータスレジスタをスタックから pop する．プログラムカウンタの値はスタック上にあり，スレッド切り替え関数から抜ける際にロードされる．

5.1.3 セマフォ

セマフォはスレッド間の排他処理を行う．開発者はアプリケーション内のクリティカルセクションをセマフォで守り，スレッドセーフなアプリケーションを開発できる．アプリケーションは割り込みを禁止することでアトミックな処理を行えるが，クリティカルセク

ション内の処理が長いと割り込み禁止の期間も長くなってしまう．セマフォは短い割り込み禁止期間でスレッド間の排他処理を実現する．

セマフォはブロックせずにウェイトを呼び出せる回数を保持する空きカウンタを持ち，セマフォを識別するユニークな ID と最新待機番号カウンタと，未解消待機番号カウンタを持つ．アプリケーションはセマフォ初期化関数とセマフォウェイト関数とセマフォリリース関数を通じてセマフォを操作する．セマフォモジュールの構成を図 5.3 に示す．

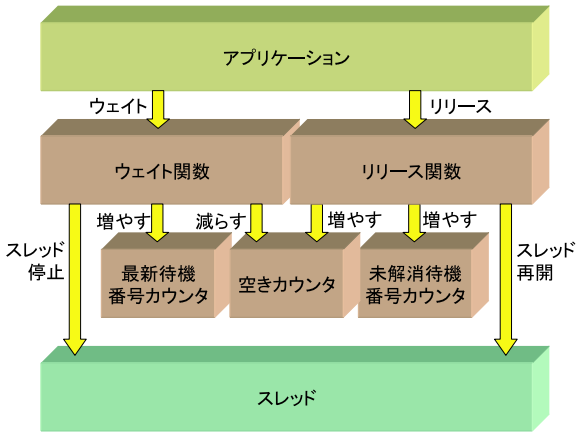

図 5.3: セマフォモジュールの構成

セマフォ初期化関数

セマフォ初期化関数はセマフォを初期化する．空きカウンタの値を引数で与えられた値に代入し，最新待機番号カウンタと未解消待機番号カウンタを 0 にリセットし，ユニークな ID をセットする．アプリケーションはセマフォウェイト関数とセマフォリリース関数を呼ぶ前にこの関数でセマフォを初期化しなくてはならない．

セマフォウェイト関数

セマフォロックは空きカウンタをチェックし，カウンタが 1 以上の場合はカウンタをデクリメントし，関数から抜ける．すでにカウンタが 0 だった場合はセマフォの ID と最新待機番号カウンタの値を引数にスレッド停止関数を呼び出し，セマフォウェイト関数を呼び出したスレッドを停止し，最新待機番号はインクリメントされる．ブロックしたスレッドはセマフォリリース関数により再開される．

セマフォリリース関数

セマフォリリース関数は最新待機番号カウンタと未解消待機番号カウンタの値を比較する．両者が同じ番号の場合はブロックしているスレッドがないので空きカウンタをインクリメントし，関数を抜ける．二つの待機番号カウンタの値が違う場合，セマフォの ID と未解消待機番号カウンタの値を引数にスレッド再開関数を呼び出すことでブロックしているスレッドを再開させる．

5.1.4 同期 I/O

同期 I/O モジュールはデバイスに対する入出力の機構を提供する．本稿では，USART の同期 I/O を例に説明をする．USART の同期 I/O モジュールには入力用モジュールと出力用モジュールがある．

入力用モジュール

入力用の同期 I/O モジュールは，リングバッファとバッファに入っているデータ数を管理するセマフォを持ち，入力初期化関数とデータ取得関数とデータ保存関数を通じて操作される．構成を図 5.4 に示す．

入力初期化関数
入力初期化関数はリングバッファとセマフォを初期化し，USART の受信完了割り込みを許可する．アプリケーションはデータ取得関数を呼び出す前にこの関数を呼び出さなくてはならない．

データ取得関数
アプリケーションはデータ取得関数を通じてシリアルのデータを取得する．データ取得関数はデータ保存関数はセマフォに対してウェイトし，リングバッファに保存されたデータを読み取る．リングバッファにデータがない場合はセマフォに対するウェイトでブロックし，データ保存関数によって再開される．

データ保存関数
データ保存関数は USART のデータ受信完了割り込みで呼び出される．シリアルからのデータを読み取り，リングバッファに格納し，セマフォをリリースする．データ取得関数内でブロックしているスレッドはセマフォのリリースで再開される．

出力用モジュール

出力用の同期 I/O モジュールはセマフォを持ち，出力初期化関数とデータ出力関数と出力可能通知関数を通じて操作される．構成を図 5.5 に示す．

出力初期化関数
出力初期化関数はセマフォを初期化し，USART の送信完了割り込みを許可する．アプリケーションはデータ出力関数を呼び出す前にこの関数を呼び出さなくてはならない．

データ出力関数
アプリケーションはデータ出力関数を通じてデータを書き込む．データ出力関数は USART が出力可能な場合，データをデバイスに出力する．出力可能でない場合は，

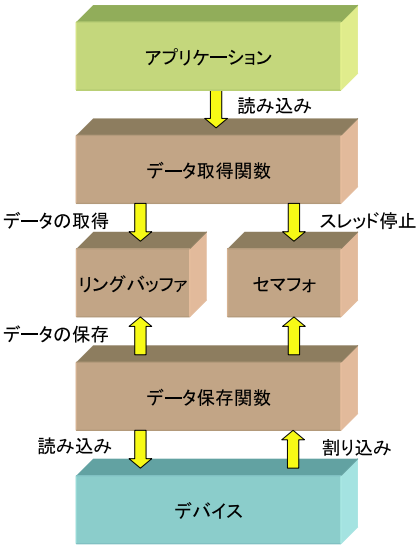

図 5.4: 入力用同期 I/O の構成

セマフォウェイト関数を呼び出し，スレッドをブロックさせる．ブロックされたスレッドは出力可能通知関数により再開され，USART にデータを出力する．

出力可能通知関数

出力可能通知関数は USART からの送信完了割り込みで呼び出され，セマフォをリリースし，データ出力関数によってブロックされているスレッドを再開させる．

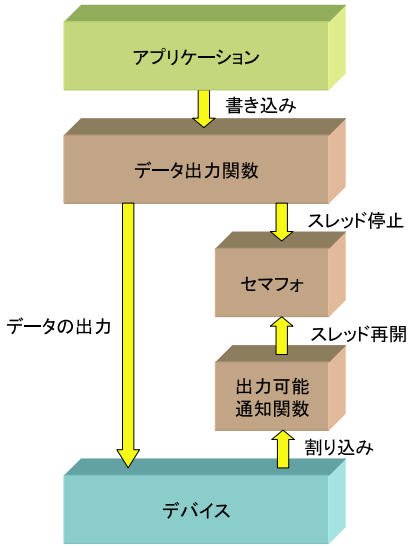

図 5.5: 出力用同期 I/O の構成

5.2 SNAOの実装

SNAOは図5.6のようなモジュールで構成されており，4.3で述べた構文解析は字句解析部，構文解析部，モデル生成部で行われる．関数の複製，利用レジスタの変更，最適化ファイルの生成は最適化部で行われる．

字句解析部はflex，構文解析部はyacc，モデル生成部と最適化部はC++で記述されている．詳しい実装環境を表5.2にまとめる．

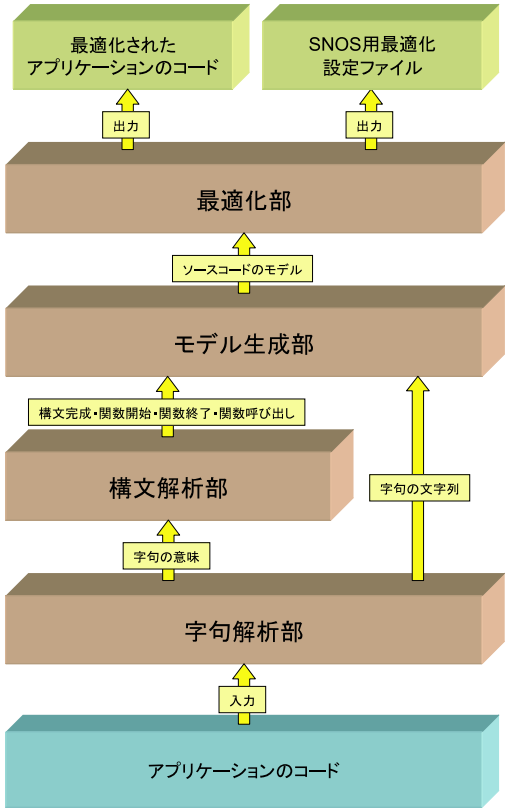

図 5.6: SNAOのモジュール構成図

5.2.1 字句解析部

字句解析部はアプリケーションのアセンブラコードを入力として受け，コードを各種レジスタ，命令，ラベル，数値，コメントなどの単位のトークンに分解する．分解されたトークンはモデル生成部に渡され，各トークンの意味は構文解析部に渡される．

表 5.2: 実装環境

項　　目	環　　境
CPU	PentiumM 1.7 GHz
Memory	512MB
字句解析	flex 2.5.4
構文解析	yacc 1.875b
コンパイラ	gcc 3.3.3(cygwin special)
ターゲットのコンパイラ	avr-gcc 3.3.2 (-O2 -S)

5.2.2　構文解析部

構文解析部は字句解析部から通知されたトークンの意味を解釈する．構文解析部が抽出するのは構文の完成，関数の始点と終点，関数内で呼ばれている関数である．図 5.7 に avr-gcc が生成するアセンブラコードの例を示す．以下にそれぞれの解析方法について述べる．

構文の完成

単一，又は複数のトークンから構成される構文の完成を調べる．図 5.7 の 1 行目のように複数のトークンによって構成される構文や，2 行目の ret 命令と 3,4 行目のコメントのように単一のトークンで構成される構文がある．構文の完成はモデル生成部に通知される．

関数の始点と終点

関数の始点は図 5.7 の 6 行目や 28 行目に見られる.global のトークンで判定し，その直後のトークンが関数名となる．関数の終点は「.size (関数名), .-(関数名)」で判定する．関数の開始，終了はモデル生成部に通知される．

関数の呼び出し

関数の呼び出しは図 5.7 の 17 行目や 20 行目に見られるプログラムカウンタからの相対アドレスによるサブルーチンコールの rcall 命令や絶対アドレスによるサブルーチンコールの call 命令とその後に続く関数名のトークンを調べる．関数呼び出しはモデル生成部に通知される．

5.2.3　モデル生成部

モデル生成部は字句解析部から受け取ったトークンを関数や構文ごとに整理する．モデル生成部が生成するモデルは図 5.8 のような構成である．以下でそれぞれの要素について述べる．

```
 1      pop r28
 2      ret
 3 /* epilogue end (size=3) */
 4 /* function main2 size 30 (23) */
 5      .size main2, .-main2
 6 .global main3
 7      .type main3, @function
 8 main3:
 9 /* prologue: frame size=0 */
10      push r28
11      push r29
12      in r28,__SP_L__
13      in r29,__SP_H__
14 /* prologue end (size=4) */
15      ldi r22,lo8(98)
16      ldi r24,lo8(43)
17      rcall sub2
18      ldi r22,lo8(9)
19      ldi r24,lo8(46)
20      rcall sub3
21 /* epilogue: frame size=0 */
22      pop r29
23      pop r28
24      ret
25 /* epilogue end (size=3) */
26 /* function main3 size 13 (6) */
27      .size main3, .-main3
28 .global main4
29      .type main4, @function
30 main4:
31 /* prologue: frame size=0 */
32      push r28
```

図 5.7: avr-gcc によって生成されたアセンブラコード

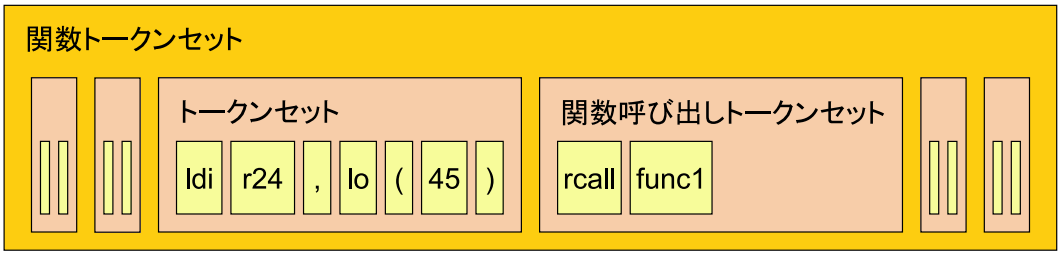

図 5.8: コードのモデル

トークン
字句解析部によって分解された最小単位．トークンはトークンセット又は関数呼び出しトークンセットに保持される．

トークンセット
構文を構成するトークンの集まり．トークンセットの区切りは構文解析部から通知される．

関数呼び出しトークンセット
トークンセットのうち，関数呼び出しを行うトークンセット．よって，最初のトークンは必ず call か rcall である．

関数トークンセット
トークンセットと関数呼び出しトークンセットを関数の単位でまとめたもの．トークンを直接保持することはない．

モデル生成部は各関数トークンセット内の関数呼び出しトークンセットを呼び出し先の関数トークンセットに結びつけることでコールグラフを生成する．

5.2.4 最適化部

最適化部は 4.3 で述べた最適化を行う．atmega128L は 8bit の汎用レジスタ r0 から r31 を持つ．レジスタ番号によって利用方法が異なり，レジスタ置き換え方法が変わる．以下にそれぞれの置き換え方法について述べる．

r0,r1
atmega128L は乗算の結果を r0,r1 に格納する．また，avr-gcc は r0 を一時的に値を格納するレジスタとして用い，r1 を値が常に 0 のレジスタとして用いる．最適化の際には r0,r1 のレジスタは置き換えない．

r2〜r15
r2〜r25 は特別な用途は決まっていない汎用レジスタである．最適化の際には r2〜r15 のレジスタは r2〜r31 の別のレジスタに置き換えられる．

r16〜r25

一部の命令はr16〜r31のレジスタのみ対応するものがある．r26〜r31はさらに特別な用途があり，最適化の方法が異なる．最適化の際はr16〜r25のレジスタはr16〜r31に置き換えられる．

r26〜r31

r26〜r31はデータ空間の間接アドレッシングに用いられる16 bitのポインタとして扱われる．r26〜r31のレジスタを二つずつ上位8ビット下位8ビットとして扱い，それぞれX,Y,Zレジスタと呼ばれる．最適化の際にはX〜Zのレジスタは X〜Zの別のレジスタに置き換えられる．

5.2.5 avr-gccによるアプリケーションの制約

atmega128Lは除算命令を持たないため，avr-gccは除算関数呼び出しで代用する．乗算を関数呼び出しに変換することがある．これらの関数はavr-gccのライブラリで提供されている．SNAOはライブラリの最適化を行えないため，アプリケーションは掛け算の処理には代替関数divide,multiplyを利用する．

5.3 本章のまとめ

本章では，SNOS及びSNAOの実装について述べた．次章では，SNOS及びSNAOの定量的評価及び定性的評価を行う．

第6章　評価

本章では，プリエンプティブマルチスレッドオペレーティングシステム SNOS と SNOS 用アプリケーションの最適化ツール SNAO について定性的評価，定量的評価を行う．

6.1 定量的評価

本節ではプリエンプティブマルチスレッドオペレーティングシステム SNOS と最適化ツール SNAO の定量的評価を行う.

6.1.1 SNOS の基本性能

SNOS のコンテキストスイッチは 250 クロックで完了する．Atmega128L は 4MHz で動作するため，62.5 マイクロ秒かかる．汎用レジスタの退避，復元は 128 クロックかかる．SNAO の最適化により，コンテキストスイッチの処理を最大 120 クロック短縮可能である.

本研究では同期 I/O として USART の出力モジュールを実装した．SNOS のコードサイズは約 2400 バイトである．各モジュールが必要とするメモリサイズを表 6.1 にまとめる．これらのモジュールが消費するメモリに加え，アプリケーションが利用する変数やスタック領域が必要となる.

表 6.1: SNOS 各モジュールの消費メモリ

モジュール	サイズ
スレッド	3 + (7 * スレッドの数) byte
セマフォ	4 * セマフォの数 byte
USART 出力	4 byte

6.1.2 SNAO の最適化速度

図 6.1 に示すコードをアセンブラに変換し，SNAO で最適化を行い，オブジェクトファイル生成にかかった時間を time コマンドで計測した．また，SNAO による最適化を行わず直接オブジェクトファイルを生成するのにかかった時間を計測した．表 6.2 に結果をまとめる．普通にコンパイルするのに比べ，SNAO による最適化を行うと約 2.5 倍の時間がかかる．SNAO なしでアプリケーションと SNOS をリンクすることは可能なので，デバッグ時などは SNAO を利用しないことで，高速なコンパイルが可能である.

6.1.3 SNAO の最適化効率

図 6.1 に示すコードを SNAO で最適化し，最適化結果をアセンブラコードのファイルサイズ，オブジェクトファイルのファイルサイズ，退避するレジスタ数で比較した．表 6.3 に結果を示す.

最適化によって退避するレジスタがなくなったが，関数の複製によりオブジェクトファ

```
/* スレッド1のエントリポイント */
void thread1(void){
  char i;
  i=1;
  i=sub1(i);
  i=sub2(i);
}

/* スレッド2のエントリポイント */
void thread2(void){
  char i;
  i=2;
  i=sub1(i);
  i=sub3(i);
}

char sub1(char i){
  return i+1;
}

char sub2(char i){
  return i+sub4(i);
}

char sub3(char i){
  return i-sub4(i);
}

char sub4(char i){
  return i+i;
}
```

図 6.1: 最適化するコード

表 6.2: SNAO の最適化速度

	user(msec)	**sys(msec)**	**real(msec)**
gcc(.c →.o)	12	10	86.8
gcc(.c →.s)	12	7	70.6
SNAO	103	25	109.5
gcc(.s →.o)	12	9	38.5

表 6.3: SNAO の最適化結果

項目	**最適化前**	**最適化後**
アセンブラコードのサイズ	1974 byte	2640 byte
オブジェクトファイルのサイズ	988 byte	1068 byte
重複したレジスタ数	3 本	0 本

イルのサイズが 80 byte 増加した．しかし 80 byte のコードサイズはターゲットのコード領域の 128KB と比べれば小さい．

6.2 定性的評価

本節では SNOS の定性的評価として，プリエンプティブマルチスレッドにより開発が容易になったかをスレッドを用いたコードとポーリングを用いたコードで比較する．例として 5 秒間隔で温度を，7 秒間隔で湿度を無線で送信するコードを示す．ポーリングを用いたコードを図 6.2 に，スレッドを用いたコードを図 6.3 に示す．

図 6.2 のように，ポーリングを用いたコードは，温度に関わるコード，湿度に関わるコード，スケジューリングに関わるコードが一つの関数に入り混じっており，わかりにくい．図 6.3 のように，スレッドを用いたコードは，温度に関わるコード，湿度に関わるコード，スケジューリングに関わるコードが関数単位で分かれており，関心事の分離がはっきりと行われている．

6.3 本章のまとめ

本章では定量評価として SNOS の基本性能を評価し，SNAO の最適化速度と最適化結果を評価した．また，定性評価として開発の容易さの評価としてマルチスレッドとポーリングのコードを比較した．次章では今後の課題を述べ，最後に本論文をまとめる．

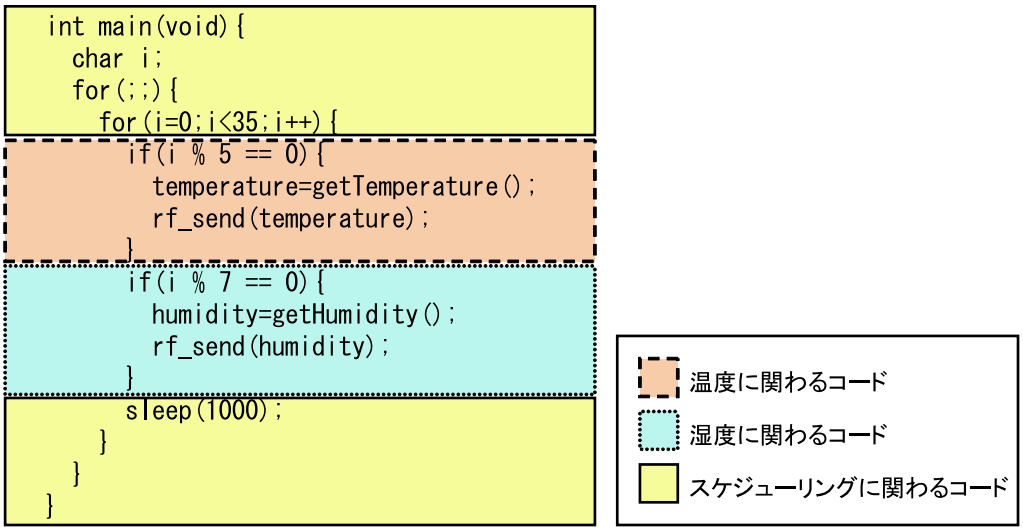

図 6.2: ポーリングによる記述

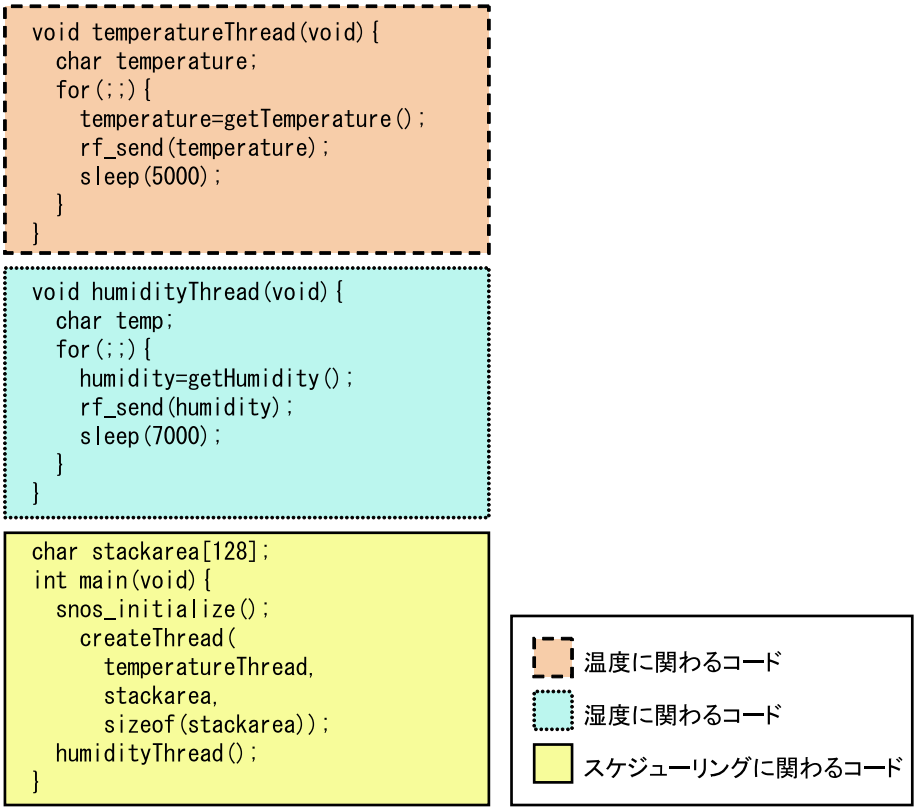

図 6.3: スレッドによる記述

第7章　結論

本論文では，センサノード用プリエンプティブマルチスレッドオペレーティングシステム SNOS とアプリケーション最適化ツール SNAO を設計し実装した．本章では今後の課題を挙げ，最後に本論文をまとめる．

7.1 今後の課題

SNOS の今後の課題として各種デバイスへの対応とハードウェアの抽象化を挙げる．また，SNAO の今後の課題としてバイナリファイルの対応と間接関数呼び出しの対応を挙げる．

- 各種デバイス対応

  本論文で SNOS が対応しているデバイスは LED と USART である．より多くのアプリケーションに対する適応のために，各種のデバイスへの対応が望まれる．

- ハードウェアの抽象化

  現状の SNOS は Mica2 mote 用に書かれている．より多くのハードウェアに対応するために抽象化の層を加える必要がある．

- バイナリファイルの対応

  本論文ではアプリケーションの最適化をアセンブリコードで行ったため，アプリケーションはライブラリファイルなどを利用できなかった．より利用し易いように，コンパイラに組み込むか，オブジェクトファイルやアプリケーションバイナリを直接書き換えるべきである．

- 間接関数呼び出しの対応

  現在の SNAO は間接関数呼び出しをトレースできないため，完全なコールグラフを生成できない．

7.2 まとめ

本論文では，センサノード用オペレーティングシステム SNOS とアプリケーション最適化ツール SNAO を設計，実装し，評価した．

ユビキタスコンピューティング環境では，ユーザや周囲の状況に即したサービス提供が行われる，サービス実現のために，実世界の情報を取得する機能を持つセンサノードが遍在し，ネットワークを介することで実世界の情報を細かに取得する．現在多くのセンサノード用アプリケーションコードはマイクロスレッドを採用した軽量なオペレーティングシステムか単純なライブラリを利用して書かれている．このような手法で開発するには，割り込みやスケジューリングなどのハードウェアに関する知識が必要であり，センサノード用アプリケーション開発が未経験の開発者には難しい．多くの開発者は PC のオペレーティングシステムが提供するプリエンプティブマルチスレッドを利用経験がある．しかしセンサノードは小型化と省電力化のために CPU やメモリなどのリソースが限られており，プリエンプティブマルチスレッド機構を搭載するにはコンテキストスイッチによるメモリや処理時間の消費が大きい．

本研究ではアプリケーションに特化した最適化により，コンテキストスイッチの処理をコンパイル時に削減可能なプリエンプティブマルチスレッドオペレーティングシステム SNOS 及び最適化ツール SNAO を構築した．本研究はプリエンプティブマルチスレッドの提供により，センサノード用アプリケーション開発が未経験の開発者でも容易に開発を行える．

謝辞

本研究の機会を与えてくださり，ご指導を賜りました慶応義塾大学環境情報学部教授徳田英幸博士に深く感謝いたします．慶應義塾大学徳田・村井・楠本・中村・南合同研究会の先輩方には折りにふれ貴重な指導と助言を頂きました．特に，徳田研究室の先生方や先輩方，ACE(Active Computing Environmets) 研究グループの方々に深く感謝いたします．また，中西健一氏，村上朝一氏，出内将夫氏には絶えざる励ましや丁寧なご指導をを賜りました．最後に，本研究を通じて様々経験や刺激を受ける機会を頂きましたことに，深く謝意を表します．

平成 16 年 12 月 29 日
須之内 雄司

参考文献

[1] Mica2 mote.
http://www.xbow.com/Products/productsdetails.aspx?sid=72.

[2] H. Abrach, S. Bhatti, J. Carlson, H.Dai, J.Rose, Seth A, B. Shucker, J. Deng, R. Han. Mantis: System support for multimodal networks of - in-situ sensors.

[3] Michael Beigl, Tobias Zimmer, Albert Krohn, Christian Decker, Philip Robinson. Smart-its - communication and sensing technology for ubicomp environments.

[4] David Gay, Philip Levis, and Robert von Behren. The nesc language: A holistic approach to networked embedded systems.

[5] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pp. 93–104, 2000.

[6] J. M. Kahn K. S. J. Pister and B. E. Boser. Smart dust: Wireless networks of millimeter-scale sensor nodes, 1999.

[7] 永原崇範, 鹿島拓也, 猿渡俊介, 川原圭博, 南正輝, 森川博之, 青山友紀, 篠田庄司. ユビキタス環境に向けたセンサネットワークアプリケーション構築支援のための開発用モジュール u^3（u-cube）の設計と実装. 電子情報通信学会技術研究報告, IN2002-243, NS2002-270, 2003.