卒 業 論 文

高水準言語のためのIoTプログラミング環境の研究

東北大学工学部
情報知能システム総合学科

B4TB2039 大塚 祐貴

指導教員　大堀 淳 教授

概要

本研究では，IoT デバイスの情報を高水準言語で扱うための仕組みの実装を行った．IoT デバイスとは，インターネットに接続したデバイスのことでそのデバイスは周囲の環境から得られる情報などを所有する．IoT デバイスが持つ情報を高水準に扱う仕組みの実装を行うことで，IoT デバイスの情報を用いたアプリケーションサービスの実装等が容易に行えると考えられる．そこで，IoT デバイスを実装し，その IoT デバイスが持つ情報を高水準言語である SML#を用いて操作する仕組みの実装を行った．

IoT デバイスの実装では，IoT デバイスのハードウェアを制御する仕組みをもつハードウェア層と，IoT デバイスの情報を高水準言語に提供する仕組みを持つプレゼンテーション層に分けて実装を行った．ハードウェア層の実装では，Raspberry Pi と GrovePi+並びに Grove センサーを組み合わせて行った．ハードウェア層全体の制御は Raspberry Pi が行うこととし，Raspberry Pi が GrovePi+を介して Grove センサーを制御し，周囲の環境情報を取得する仕組みを実装した．プレゼンテーション層では，Raspberry Pi から情報を送信するサーバー側とその情報を受信し高水準言語に渡すクライアント側に分けて実装を行った．サーバー側とクライアント側の通信には HTTP を用いて行った．そのために，Raspberry Pi に Web サーバーである Apache2 を導入し，CGI プログラムを実行することでクライアント側に JSON 形式の IoT デバイスが持つデータを送信する仕組みの実装を行った．また，SML#の C 言語との連携機能を用いて TCP 通信を行い，サーバー側から JSON データの受信を行い，そのデータを SML#の JSON サポート機能を用いて認識させ，IoT デバイスの情報を操作する機能の実装を行った．

本論文では，IoT デバイスの実装に必要な Raspberry Pi と GrovePi+並びに Grove センサーの詳細な情報を述べた後，IoT デバイスの実装の詳細とその性能について述べる．

目 次

第1章　序論

1.1　研究の背景と目的

現在，従来までネットワーク機能を必要としていなかった様々な「モノ」が，インターネットに接続されるようになった．このようにインターネットに接続している様々な「モノ」全般を,「IoT（Internet of Things）」と呼ぶ [2]．様々な「モノ」をインターネットに接続するメリットの１つに，その「モノ」の持つ情報をインターネット上で共有できることが挙げられる．そうすることで，インターネット経由で情報を受け取ることができるデバイス上で，その「モノ」単体では実現できなかったアプリケーションの作成が行える．しかし，IoT を用いたサービスの多くは，クラウド上にあるアプリケーションサービスで「モノ」から得られる情報を処理し，ユーザーはその結果を受け取る，といったものであり，インターネットに接続している「モノ」を高水準に扱う環境が整っていないと考えられる．

一方，コンピュータに導入されたオペレーティングシステム（OS) は，コンピュータに接続されたデバイスを，高水準言語で操作できるような環境を持っている．そこで，本研究の最終目標を，インターネットに接続している「モノ」（以下 IoT デバイスと呼ぶ）を一般にコンピュータに接続しているハードウェアと同様に扱う，IoT デバイスのためのオペレーティングシステム（IoT OS）の実現とする．IoT OS が実現すると，ネットワークに存在する IoT デバイスを検出し，検出された IoT デバイス全ての情報を取得するための初期化を行い，常時変化するであろう IoT デバイスが持つ情報を取得し，その情報をユーザーに提供する仕組みを得ることができる．この目標を達成するために，まずは IoT デバイスのハードウェアを制御する仕組みと，その IoT デバイスが持つ情報を高水準言語に見せるための仕組みの実装を行う．ここでは，前者をハードウェア層と呼び，後者をプレゼンテーション層と呼ぶこととする．

IoT デバイスのハードウェア層は，Linux の機能を持ちセンサー等の周辺機器デバイスの操作が容易な Raspberry Pi と，Raspberry Pi に接続可能で周囲の環境情報を取得できるハードウェアをもつ GrovePi+及び Grove センサーとを組み合わせて構成する．Raspberry Pi を用いて IoT デバイスのハードウェア全体を制御し，GrovePi+及び Grove センサーを用いて IoT デバイスの周囲の環境情報を取得する．Raspberry Pi と GrovePi+及び Grove センサーを IoT デバイスのハードウェアとして組み合わせて使用するためには，Raspberry Pi と GrovePi+それぞれの詳細な仕様や操作方法等の知識が必要である．そこで，本論文では，Raspberry Pi と GrovePi+及び Grove センサーそれぞれについて，IoT デバイスのハードウェア層の構築に必要な知識をまとめ報告する．

また，IoT デバイスのプレゼンテーション層には，IoT デバイスの情報を取得し送信を行うサーバー側と，サーバー側から送信される IoT デバイスの情報を処理し高水準言語に渡すクライアント側に分けて実装した．サーバー側とクライアント側との通信には，HTTP を用いた通信を導入した．HTTP 通信を行うネットワークについても構築し，そのネットワークは安全のためインターネットには接続しないものとした．

以上の導入により，IoT デバイスのハードウェア層とプレゼンテーション層の実装を行うことができ，IoT デバイスが接続するネットワーク内で HTTP を用いて IoT デバイスが持つ情報を正常に確認

できた．本論文では，IoT デバイスのハードウェア層の構築に必要である Raspberry Pi と GrovePi+ 及び Grove センサーの詳細な情報についてまとめた後，実際にハードウェア層とプレゼンテーション層の実装方法について説明する．

1.2 本論文の構成

本論文の構成は以下の通りである．第 2 章では，Raspberry Pi を用いて IoT デバイスのハードウェア層を構成するために必要な情報を説明し，Raspberry Pi に接続可能な周辺機器の操作方法について述べる．第 3 章では，Raspberry Pi と同様に IoT デバイスのハードウェア層を構成する GrovePi+ についての必要な情報を説明し，その仕様と操作方法について述べる．第 4 章では，Raspberry Pi と GrovePi+で構成される IoT デバイスの概要を述べ，ハードウェア層の実装について説明する．第 5 章では，IoT デバイスのプレゼンテーション層の実装について説明し，IoT デバイスが持つ情報を高水準言語を用いて扱う方法を述べる．第 6 章では，本研究の現状のまとめと，今後の展望について述べる．

第2章　Raspberry Piについて

本章では，IoT デバイスのハードウェア層を実装するために必要な Raspberry Pi の情報について，まとめ報告する．まずは，2.1 で Raspberry Pi の概要について述べ，2.2 で本研究で用いた Raspberry Pi の仕様について述べる．その後，2.3 では，本研究で用いる Raspberry Pi に最低限必要な動作環境の構築方法について説明し，2.4 では IoT デバイスのハードウェア層を実装するために欠かせない，Raspberry Pi を用いて周辺機器を操作する方法について詳細に述べる．

2.1　Raspberry Piの概要

Raspberry Pi は，Raspberry Pi 財団が開発している小型のシングルボードコンピュータである [2]．Raspberry Pi のハードウェア的な特徴として，USB，HDMI，LAN それぞれのポートを持つことが挙げられる．これによって，キーボードやディスプレイを接続することができ，デスクトップコンピュータとして使用することが可能であるし，インターネットに接続することもできる．また，Raspberry Pi は，GPIO（General Purpose Input/Output：汎用入出力）と呼ばれるピンを複数本持っている．この端子は周辺機器接続端子として機能し，GPIO に接続する他のデバイスに対し，操作を受け付けたり操作を行ったりすることができる．Raspberry Pi 財団は，Raspberry Pi のためのオペレーティングシステムとして，Debian GNU/Linux に基づいた Linux OS である「Raspbian（ラズビアン）」を提供している．本研究では，Raspberry Pi に Raspbian を導入し，その Raspberry Pi を用いて IoT デバイスのハードウェア層を実装した．

2.2　Raspberry Piの仕様

Raspberry Pi は，搭載される機能によって様々なモデルに分類されている．本研究で用いた Raspberry Pi は，Raspberry Pi 3 Model B である．この先，本論文中で Raspberry Pi と表記してある場合，それは全て Raspberry Pi 3 Model B のこととする．本研究で用いた Raspberry Pi 3 Model B の主な仕様 [2] は，表 2.1 のとおりである．

次に，Raspberry Pi の外部インターフェイスの 1 つである周辺機器接続端子について説明する．Raspberry Pi の基板には 40 ピンの周辺機器接続端子がある．ピンの種類は 5 種類あり，40 ピン全てにそれぞれ種類が割り当てられている．存在するピンの種類は，5V 電源端子，3.3V 電源端子，GND（接地）端子，ID EEPROM 端子，GPIO 端子，の 5 種類となっている．種類ごとのピンの本数は，5V 電源端子ピンが 2 本，3.3V 電源端子ピンが 2 本，GND 端子ピンが 8 本，ID EEPROM ピンが 2 本，GPIO ピンが 26 本となっている．周辺機器端子のピン配置を，図 2.1 に示す．図 2.1 中の右下の図が，Raspberry Pi 3 Model B の基板の模式図で，上の図が，Raspberry Pi 3 Model B の基板に付いている周辺機器接続端子の拡大図である．ここで，周辺機器接続端子の拡大図中の丸の種類について説明する．丸 1 つはピン 1 本を意味する．黒塗りのピンは，GND ピンである．5V と書かれたピンは，5V

表 2.1: Raspberry Pi 3 Model B の主な仕様

プロセッサ	BCM2837
CPU	ARM v8 64bit 1.2GHz 4 コア
メモリ	1GB
周辺機器接続端子 ピン数	40（GPIO 端子は 26 ピン）
有線 LAN	10/100Base
無線 LAN	IEEE 802.11 b/g/n 2.4GHz
USB2.0 ポート	4 つ
ストレージ	microSD カード
推奨電源容量	5.0V 2.5A 12.5W
電源ソース	microUSB
質量	45g
大きさ	85.6mm × 56.5mm

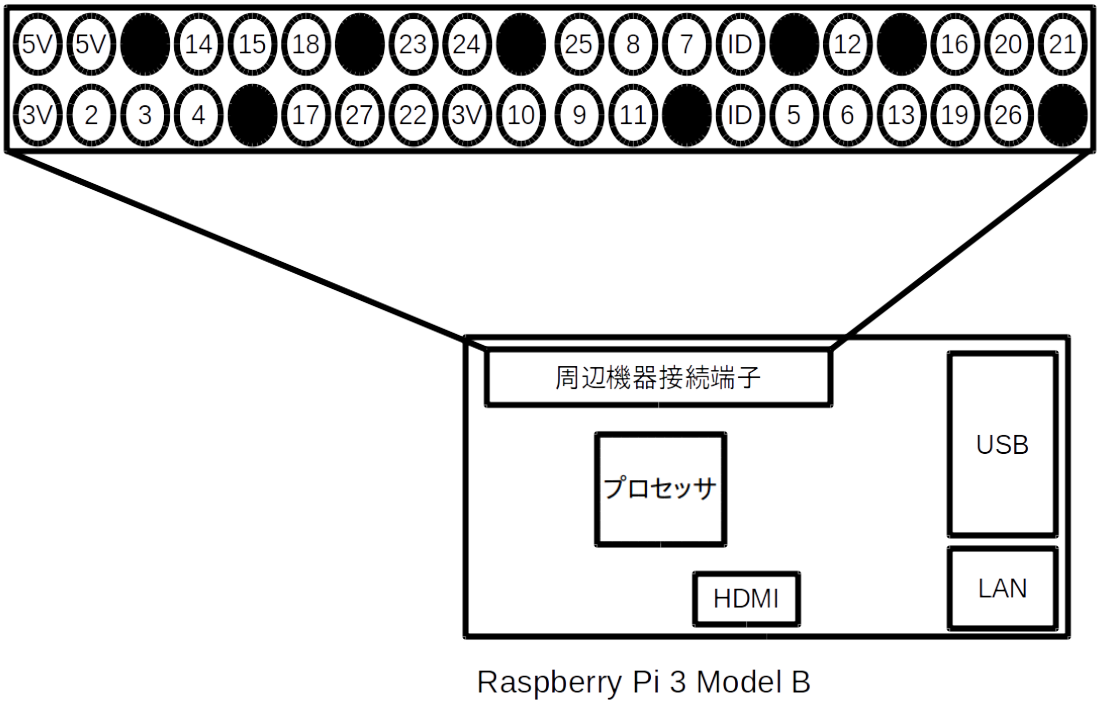


図 2.1: Raspberry Pi 3 Model B の基板の模式図と周辺機器接続端子

電源ピンである．3V と書かれたピンは，3.3V 電源ピンである．ID と書かれたピンは，ID EEPROM ピンである．それ以外で数字が書かれたピンは，GPIO ピンである．GPIO ピンそれぞれに固有の番号が振られており，その番号が図中の数字と対応する．

次に，GPIO ピンについて説明する．GPIO ピンは，全てのピンに対し入力または出力のいずれかに設定することができる．出力の設定にしたピンは，Raspberry Pi の操作によって，端子の電圧を 0V か 3.3V にすることができる．入力の設定にしたピンは，Raspberry Pi の操作によってその端子にかかる電圧の値に応じたデジタル値を取得できる．入力に設定されたピンにかかる電圧が 0V の時にはデジタル値 0 が，ピンにかかる電圧が 3.3V の時にはデジタル値 1 が，それぞれ取得できる．また，GPIO の特定のピンに関してのみ，Raspberry Pi の操作によって GPIO とは異なる機能を割り当てることが可能である．その機能の一部が，I^2C（Inter-Integrated Circuit）によるシリアル通信機能である．本研究では，Raspberry Pi から GPIO を操作する機会がなかったため，GPIO の操作方法については説明しない．また，I^2C 通信に関しては，IoT デバイスのハードウェア層を実装する際に用いるため，I^2C について 2.4.1 で説明し，Raspberry Pi を用いた I^2C の操作方法については 2.4.2 で述べる．

2.3 Raspberry Pi の動作環境の構築

IoT デバイスのハードウェア層を実装するために，まずは最低限の動作環境が整った Raspberry Pi が必要である．ここで，最低限の動作環境とは，Raspberry Pi のストレージにインストールされた Raspbian が起動し，Raspberry Pi がネットワークに接続することができる状態にある環境のことである．そのために必要なものは，Raspberry Pi 本体，電源アダプタと microUSB ケーブル，microSD カード，LAN ケーブルである．Raspberry Pi の電源端子は microUSB 端子のため，microUSB ケーブルと USB 端子用の電源アダプタを用いて Raspberry Pi に電源を供給する．また，Raspberry Pi は OS 用の記憶領域を持たず，ストレージとして microSD カードを使用する．そのため，予め OS をインストールした microSD カードを用意しておく必要がある．2.3.1 で，microSD カードに OS（Raspbian）を書き込み，その microSD カードを使用して Raspberry Pi を起動させるまでの流れを説明する．また，起動後の Raspberry Pi をネットワークに接続する方法については，2.3.2 で説明する．

2.3.1 OS のインストールと Raspberry Pi の起動

本研究では，Raspberry Pi 財団が提供している，Raspberry Pi 標準の OS である Raspbian を使用した．Raspbian は，Debian GNU/Linux をベースにした Raspberry Pi 標準の OS である．そのため，Raspbian は Raspberry Pi に最適化がなされており，Raspberry Pi の各種 IO を操作するためのドライバや設定ツールが予め Raspbian に含まれている．

Raspbian は，Raspberry Pi の公式サイト [3] からダウンロードできる．Raspbian には，通常版と LITE 版の 2 種類ある．通常版はデスクトップ環境などの GUI が用意されたもので，LITE 版は GUI が用意されていない．どちらを選んでも OS のインストール方法は同じであり，本研究では GUI は使用しないため LITE 版を使用した．

Raspbian のバージョンは，Debian のバージョンに応じて，日々更新されている．本研究を行った 2017 年 11 月時点での Raspbian のバージョンは，Raspbian Stretch であったため，本研究で導入した Raspbian は Raspbian Stretch LITE である．本論文中で，この先 Raspbian と表記してある場合，それは全て Raspbian Stretch LITE のことである．

以下に，ダウンロードしたRaspbianのイメージファイルを実際に用意したmicroSDカードに書き込むまでの手順を示す．なお，本研究で用いたmicroSDカードの容量は8GBである．また，Windowsが導入されたPCを用いて書き込みを行った．

1. Sourforgeのプロジェクトページ [2] から，Win32 Disk Imagerのインストーラーをダウンロードし，Windowsにインストールする．ここで，Win32 Disk Imagerはイメージファイルをディスクに書き込む機能を持つソフトウェアのことである．
2. microSDカードをカードリーダに挿入し，Windows PCへ接続する．
3. Win32 Disk Imagerを実行する．
4. Win32 Disk ImagerのImage Fileという欄に，ダウンロードしたRaspbianのイメージファイルを指定する．
5. Win32 Disk ImagerのDeviceという欄で，microSDのカードリーダが接続されたドライブを選択する．
6. Win32 Disk ImagerのWriteボタンをクリックし，microSDに書き込みを行う．書き込み終了後，Win32 Disk Imagerを終了し，microSDカードを取り出す．

これで，Raspbianが書き込まれたmicroSDカードの準備ができた．

次に，Raspbianが書き込まれたmicroSDカードを用いて，Raspberry Piの起動方法について説明する．Raspberry PiにRaspbianが書き込まれたmicroSDカードを挿入し，電源用のmicroUSBをRaspberry Piに接続する．この時，Raspberry Piの基板に付いている赤色LEDが点灯したら，Raspberry Piが起動したこととなる．キーボードやモニターをRaspberry Piに接続する場合は，Raspberry Piを起動する前に，Raspberry Piに予め接続する必要がある．

2.3.2 Raspberry Piのネットワーク接続

ここでは，Raspberry Piをネットワークに接続するまでの方法と，Raspberry Piに対し固定のIPアドレスを設定する方法について説明する．

Raspbianの初期設定では，DHCPによりIPアドレスを取得することになっている．そのため，LANケーブルを用いてネットワークに接続した状態でRaspberry Piを起動すると，IPアドレスが自動で割り当てられるため，ネットワークに接続できる．

次に，無線LANを使用してネットワークに接続する方法を説明する．無線LANを用いて任意のアクセスポイントへ接続するには，そのアクセスポイントのSSIDとパスワードを事前にRaspberry Piに設定しておく必要がある．ここでは，CUIを用いてRaspberry Piにアクセスポイントの SSIDとパスワードを設定する方法のみ説明し，GUIを使用した設定方法については省略する．なお，この作業はモニターとキーボードを用いる．以下に，モニターとキーボードを接続したRaspberry Piを起動した直後の状態からの，設定方法の順序を示す．ここで，接続するアクセスポイントのSSIDを`SSID`，そのパスワードを`PASS`とする．

1. Raspberry PiのWi-fiの国コードを日本に設定する
   (a) Raspberry Piのターミナルで，`sudo raspi-config`を実行
   (b) 上下キーを用いて，`4 Localisation Option`を選択し，エンターを押す

(c) 上と同様の方法で，I4 change Wi-fi Country を選択

(d) 上と同様の方法で，JP japan を選択

2. Raspberry Pi の無線 LAN のインターフェイスを確認

(a) Raspberry Pi のターミナルで，ip addr コマンドを実行

(b) wlan0 という名前のインターフェイスがあることを確認する

3. wlan0 が接続するアクセスポイントの設定

(a) Raspberry Pi の/etc/wpa_supplicant/wpa_supplicant.conf に以下のコマンドでアクセスポイントの情報（SSID とパスワード）を書き込み

```
sudo sh -c 'wpa_passphrase SSID PASS /etc/wpa_supplicant/wpa_supplicant.conf'
```

(b) /etc/wpa_supplicant/wpa_supplicant.conf を開くと以下のような記述がある．

```
country=JP
ctrl\_interface=DIR=/var/run/wpa\_supplicant GROUP=netdev
update\_config=1
network={
ssid="SSID"
psk="PASS"
psk=cd7eb826fc6e4f7b9573358bc56a1d22770cc444d305fbe4a66ce16a99ca24fd
}
```

この中の psk="PASS"という行について，そこにはアクセスポイントのパスワードが直接記述されるため，削除する．

4. Raspberry Pi が起動後，自動で上で設定したアクセスポイントに接続するよう設定する

(a) Raspberry Pi の/etc/network/interfaces に以下を書き込む．

```
auto wlan0
allow-hotplug wlan0
iface wlan0 inet manual
wpa-conf /etc/wpa_supplicant/wpa_supplicant.conf
```

以上の操作を実行することで，Raspberry Pi は起動時に自動で無線 LAN を用いて，指定したアクセスポイントに接続でき，無線 LAN を用いてネットワークに接続できる．

次に，Raspberry Pi の IP アドレスの固定方法について説明する．Raspberry Pi の IP アドレスを固定するには，/etc/dhcpcd.conf というファイルに以下の様な記述を追加し，その後 Raspberry Pi を再起動することで行える．

```
interface wlan0
static ip_address=192.168.11.23/24
static routers=192.168.11.1
static domain_name_servers=192.168.11.1
```

この追加する記述の例では，wlan0 というインターフェイスに対し，static ip_address=の先に記述された IP アドレスを割り当てる．その下の記述は，ネットワークのルーターに関する記述である．

2.4 Raspberry Piの周辺機器の利用

Raspberry Pi は，USB ポート，HDMI ポート，LAN ポート，周辺機器接続端子などを用いて様々な周辺機器を利用できる．その中で本研究で用いたものは，周辺機器接続端子の特定の GPIO を用いた I^2C によるシリアル通信を行う周辺機器である．まずはシリアル通信規格である I^2C について述べたあと，Raspberry Pi で I^2C を用いたシリアル通信を行う方法について説明する．

2.4.1 I^2C 通信について

I^2C はシリアル通信の規格であり，シリアルクロックライン（SCL）とシリアルデータライン（SDA)の 2 本のバスラインで構成される．I^2C では，1 体 1 の通信だけでなく，1 体多の通信も可能である．

I^2C 通信は，マスターとスレーブの関係で動作する．全ての I^2C 通信において，マスターとなる 1 つのデバイスが，通信開始制御，通信を行うスレーブの選択，データの送受信の指定を行う．スレーブとなるデバイスは，同一バス上に複数個接続でき，それらは全てマスターからの要求に従い動作する．スレーブは，製造時にデバイスごとに割り振られたアドレスを持ち，マスターはそのアドレスを用いてスレーブを区別する．スレーブの持つ固有のアドレスは，7 ビットの値であり，基本的にその値を変更することはできない．ただし，一部のデバイスは，基板上の回路を物理的に書き換えることで，アドレスが変更できるものもある．7 ビットのアドレスであるため，使用可能な値は 16 進数で 0x00 から 0x7f の 128 個である．しかし，一部の値は I^2C の規格により予約されているため，実際にデバイスが持つことのできる値は 16 進数で 0x03 から 0x77 の 117 個である．

次に，I^2C の通信方法について説明する．通信は，SCL と SDA を流れる値の状態である High と Low の組み合わせによって行う．ここで，SCL の値は常に同じ周期で High と Low が交互に行き来する．SCL が High の時の SDA の値の変化によって 4 つの状態があり，それらを以下に示す．

- SCL が High の間，SDA が High から Low に切り替わる場合，この状態のことをスタートコンディションという．
- SCL が High の間，SDA が Low から High に切り替わる場合，この状態のことをストップコンディションという．
- SCL が High の間，SDA が常に High である場合，この状態は定数 1 を表す．
- SCL が High の間，SDA が常に Low である場合，この状態は定数 0 を表す．

スタートコンディションとは，マスターがスレーブに対し通信を開始する際にとる状態のことであり，ストップコンディションとは，マスターが通信を終了する際にとる状態のことである．したがって，I^2C 通信は，マスターがスタートコンディションを送信すると開始され，マスターがストップコンディションを送信すると終了する．

次に，マスターがスタートコンディションを送信してから，通信対象のスレーブと通信を行うまでの流れを説明する．通信でやり取りされるデータのサイズは 8 ビットが基本であり，そのデータに対し受信が成功したデバイスは 1 ビットのデータを送信する．マスターは，スタートコンディションを送信後すぐに，通信対象であるスレーブのアドレスデータと，送信か受信かを表す 1 ビットデータの合計 8 ビットデータ送信する．そのデータ構造は，上位 7 ビットが通信対象のスレーブのアドレスであり，下位 1 ビットが送受信を表すビットである．マスターがスレーブに対しデータを送信する際は，送受信を表すビットに 1 を，マスターがスレーブからデータを受信する際は，送受信を表すビットに

0 をそれぞれ選択する．通信を行うバス上にある全てのスレーブは，マスターから送信されるスタートコンディションと，その後に送信されるアドレスと送受信を表す 8 ビットデータを受信する．そしてスレーブは，受信したデータにあるアドレスと自身が持つアドレスを比較し，一致した場合は，マスターに対し 1 ビットのデータ 0 を送信する．マスターは，アドレス等のデータを送信した後で 0 を受信できたなら，そこから指定したアドレスを持つスレーブとデータのやり取りを開始する．

マスターがデータを受信する場合は，スレーブから 8 ビットずつデータが送信され，マスターが 8 ビットのデータを受信するたびに 1 ビットのデータをスレーブに返す．ここで，マスターが 1 ビットデータ 0 を送信しスレーブが受信した場合，スレーブはデータ送信を続行しマスターはデータを更に受信し続ける．マスターが 1 ビットデータ 1 を送信しスレーブが受信した場合，スレーブはデータの送信を終了する．

マスターがデータを送信する場合は，マスターがスレーブに 8 ビットずつデータを送信し，スレーブは 8 ビットのデータを受信するたびに 1 ビットのデータ 0 をマスターに送信する．マスターが 1 ビットデータ 0 を受信すると，再び 8 ビットデータをスレーブに送信するか，ストップコンディションを送信し通信を終了するか選択できる．

マスターがストップコンディションを送信するタイミングは，データを受信していた場合は 1 ビットデータ 1 を送信したあと，データを送信していた場合は最後のデータを送信し終えスレーブから 1 ビットデータ 0 を受信したあと，のいずれかである．しかしマスターは，これらのタイミングでストップコンディションを送信せずに，再びスタートコンディションを送信でき，この状態のことをリピートスタートコンディションという．この状態で，SCL の状態が Low の間に SDA の状態を High にし，SCL が High になったあと再び Low に変わるまでの間に SDA を Low にした場合に，リピートスタートコンディションが送信されることとなる．マスターがリピートスタートコンディションを送信した後は，マスターがスタートコンディションを送信した後と同様に通信が行われる．

マスターがスレーブのアドレスと送受信ビットを含む 8 ビットのデータを送信してから，対象のスレーブとデータの送受信を行う際，そのやり取りを行えるデータのサイズは最大で 32 バイトである．つまり，送受信は 32 回までしか行うことはできない．それ以上のサイズのデータの送受信を行いたい場合は，リピートスタートコンディションを送信し再びスレーブと送受信の選択を行うか，ストップコンディションと再度スタートコンディションを送信し，通信を新たに始めるかのどちらかを選択する必要がある．

2.4.2 Raspberry Pi の I^2C 通信について

ここでは，Raspberry Pi を用いて I^2C 通信を行う方法について説明する．Raspberry Pi はマスターとして機能し，GPIO の 2 番ピンと 3 番ピンがそれぞれ SDA と SCL の役割を持っている．

Raspberry Pi で I^2C を用いてシリアル通信を行うためには，GPIO の 2 番ピンと 3 番ピンに対し，I^2C のドライバを有効にする設定を行う必要がある．そのドライバは，Raspbian にすでに含まれており，以下の操作を行うことでドライバを有効にできる．

1. Raspberry Pi のターミナルで，`sudo raspi-config` を実行
2. 上下キーを用いて，`5 Interfacing Option` を選択し，エンターを押す
3. 上と同様の方法で，`P5 I2C` を選択
4. 確認画面が出るので，`Yes` を選択

5. Raspberry Pi を再起動する

6. ターミナルでコマンド `lsmod` を実行し，出力された結果の中に `i2c_dev` と `i2c_bcm2708` の表示があれば，I^2C が有効化された

次に，Raspberry Pi のコマンドラインから I^2C を用いた通信を行う方法を説明する．Linix で I^2C を扱うためのコマンド群である「i2c-tools」が存在するので，このコマンド群を用いると，コンソールで I^2C を用いた通信を行うことができる．このコマンド群は予め Raspbian に含まれていないため，パッケージ管理ソフトウェア等を利用しインストールする必要がある．`apt-get install i2c-tools` コマンドを実行するなどし，インストールできる．

まずは，Raspberry Pi の特定のバスに接続されているスレーブのアドレスの一覧を取得するコマンドである，`i2cdetect` について説明する．このコマンドを用いて得られるアドレスは，16 進数表記である．このコマンドは I^2C バスの番号を指定して実行する．使用可能なバスの番号は，Raspberry Pi によって決められており，このコマンドを用いて使用できるバスの番号を確認できる．このコマンドにオプション`-l` をつけた，`sudo i2cdetect -l` を実行することで，使用できるバスの番号がわかる．コマンドを実行した結果，本研究で使用した Raspberry Pi の I^2C バスの番号は 1 であることがわかった．またこのコマンドは，オプションで実行時の確認を行わないようにすることもできる．実際に Raspberry Pi のコマンドラインで，実行時の確認を行わないようにオプションをつけ実行した結果は，以下のようになる．

```
$ sudo i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          03 04 -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- 3e --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- 62 -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

この実行例では，Raspberry Pi に 0x03，0x04，0x3e，0x62 というアドレスをそれぞれ持つ 4 つのスレーブが接続されていることを意味する．

i2c-tools には，他にも指定したアドレスを持つスレーブに対し指定したデータ列を送信するコマンドである `i2cset` や，指定したアドレスを持つスレーブから指定したサイズのデータを受信するコマンドである `i2cget` 等が含まれているが，本論文では用いないため説明を省略する．

次に，コマンドを用いずに Raspberry Pi で I^2C を用いた通信を行う方法について説明する．Linux kernel に関するドキュメントの I^2C デバイスインターフェイスの説明 [4] によると，特定のデバイスファイルについて，システムコールを用いた書き込みや読み込みを行うことで，Raspberry Pi は I^2C を用いた通信を行うことができる．Linux のデバイスファイルは，全て`/dev/`内に格納されている．I^2C を用いた通信を行うデバイスファイルは，i2c-0，i2c-1，$\cdots$，というように複数あるが，名前の数字はバスの番号と対応している．そのため，本研究で用いた Raspberry Pi で I^2C を用いた通信を行うには，`/dev/i2c-1` を操作する必要がある．

デバイスファイルの操作は，C 言語を用いてシステムコールを用いたプログラムを作成し，そのプログラムを実行することで行う．プログラムの流れは以下に示す．

1. `open` 関数を用いてデバイスファイルを開き，ファイルディスクリプタを得る

2. ioctl 関数を用いてスレーブのアドレスを指定し，通信を行うスレーブを決める

3. write 関数や read 関数を使用し，デバイスファイルを読み書きすることで，送受信を行う

以上の内容を実装した C 言語によるプログラムの擬似ソースコードを以下に示す.

```
#include <linux/i2c-dev.h>
int file;
char filename = "/dev/i2c-1";

file = open(filename, O_RDWR);
if(file < 0){/*エラー処理*/}

int addr = 0x40;
if(ioctl(file, I2C_SLAVE, addr) < 0){/*エラー処理*/}

unsigned char buf[10];

if(write(file, buf, 3) != 3){/*エラー処理*/}
if(read(file, buf, 1) != 1){/*エラー処理*/}

close(file);
```

上記の擬似ソースコードの例では，バス番号 1 に接続し 0x40 アドレスを持ったスレーブと Raspberry Pi が I^2C を用いた通信を行い，データの送受信を行っている．open 関数は，引数に開くデバイスファイルのパスとモードをとる．ここで O_RDWR は読み書き可という意味のモードである．そのため，例ではデバイスファイル/dev/i2c-1 を，読み書き可モードで開いている．また，この関数の返り値は，指定されたファイルを開くことに成功した場合，0 以上の整数値であるファイルディスクリプタとなり，ファイルを開くことに失敗した場合，-1 となる．ioctl 関数は，デバイスドライバのパラメータを変更することができる．引数にオープンされたデバイスドライバのファイルディスクリプタと，その他デバイスドライバごとに異なるパラメータをとる．例では，オプションに I2C_SLAVE とスレーブのアドレスを指定している．ここで，I2C_SLAVE は，通信対象のスレーブのアドレスを変更するためのオプションである．この関数は，処理が失敗した場合，返り値が-1 となる．write 関数や read 関数は，ファイルディスクリプタが指すファイルに対して，指定したバイト分，データを読み書きする関数である．返り値は，どちらの関数も実際に読み書きを行ったバイト数である．通信を終了する際は，close 関数を用いてデバイスファイルを閉じることで，通信を終了することができる．

第3章　GrovePi+とGroveセンサーについて

本章では，Raspberry Piと一緒にIoTデバイスのハードウェア層を構成するGrovePi+ 並びにGroveセンサーの詳細を説明する．IoTデバイスのハードウェア層では，Raspberry Piを用いてGrovePi+とGroveセンサーをコントロールする必要がある．そのため，まずは3.1でGroveシステム [5] とGrovePi+の概要を説明し，3.2でGrovePi+とGroveセンサーの仕様について述べる．そして，3.3ではRaspberry PiでGrovePi+を操作するための方法について詳細に述べる．

3.1　Groveシステムと GrovePi+の概要

Groveシステム [5] とは，SeeedStudioによって開発された，モジュールを標準化したコネクタと専用のケーブルで接続する仕組みのことである．その標準化されたコネクタのことをGroveコネクタと呼び，Groveコネクタを搭載したセンサー等のモジュールのことをGroveモジュールと呼ぶ．また，Groveコネクタに接続できる専用ケーブルのことを，Groveケーブルと呼ぶ．Groveケーブルを Groveコネクタに接続するには向きが決まっており，ある1方向からのみでしか接続することができないようになっている．本論文では，Groveモジュールの中でも，周囲の環境情報を取得するセンサーが搭載されているモジュールのことをGroveセンサーと呼ぶこととする．

GrovePi+とは，Groveコネクタを多数搭載した基板のことで，Raspberry Piの周辺機器接続端子に直接接続できるコネクタを，基板の下部に搭載している．そのため，GrovePi+をRaspberry Piの上にのせるようにして接続する．GrovePi+を用いることで，Raspberry PiからGroveセンサーを使うことが可能になる．GrovePi+はDexter Industriesが開発したもので，Dexter IndustriesはRaspberry PiでGrovePi+を操作するために必要なセットアップツール等を配布している．

本研究では，GrovePi+ Starter Kit for Raspberry Piという GrovePi+，複数個の Groveモジュール，複数本の Groveケーブル，ユーザーマニュアルが付属した製品を用いて，IoTデバイスのハードウェア層の実装を行った．IoTデバイスのハードウェア層を構成する GrovePi+と Groveモジュールについて説明するために，まずは3.2.1で Groveコネクタと Groveセンサーの仕様について説明する．そのあとで，3.2.2で GrovePi+の仕様について説明する．

3.2　GrovePi+並びに Groveセンサーの仕様

3.2.1　Groveコネクタと Groveモジュールの仕様

Groveモジュール [5] には，必ず Groveコネクタが搭載されている．Groveモジュールは搭載する回路によって複数のタイプに分類されるが，Groveコネクタは全ての Groveモジュールで統一されている．

はじめに，Groveコネクタ [5] の仕様について説明する．Groveコネクタは，4つのピンから構成され，全てのピンがそれぞれで異なる機能を持っているが，そのピンごとの機能は固定である．Grove

コネクタの 4 つのピンには，ピン 1，ピン 2，ピン 3，ピン 4 という名前がそれぞれにつけられている．Grove コネクタ同士は Grove ケーブルを用いて互いを接続するが，Grove ケーブルと Grove コネクタの接続には向きが決まっているため，Grove コネクタ同士を接続したときは同じ名前をもつピン同士が接続されることとなる．また，Grove ケーブルは 4 本の導線で構成されており，4 本全ての導線で異なる色をしている．その色は Grove コネクタのピンの名前と対応しており，ピン 1 には黄色，ピン 2 には白色，ピン 3 には赤色，ピン 4 には黒色の導線がそれぞれのピンと接続する．

次に Grove モジュールのタイプ別のピンの機能について説明するために，Grove モジュールのタイプの一覧を以下に示す．

- デジタルセンサーが搭載されたモジュール
- アナログセンサーが搭載されたモジュール
- GroveUART モジュール
- GroveI2C モジュール

ここで，デジタルセンサーとは，周囲の環境情報をデジタル値として取り出すセンサーのことであり，アナログセンサーとは，周囲の環境情報をアナログ値として取り出すセンサーである．ここでは，前者のモジュールをデジタルセンサー，後者のモジュールをアナログセンサーと呼ぶこととする．また，GroveUART モジュールと GroveI2C モジュールについては，本研究では使用しないため説明を省略する．

次に，アナログセンサーとデジタルセンサーそれぞれの，Grove コネクタのピンについて説明する．両モジュールの Grove コネクタのピン 3 とピン 4 は，それぞれ Grove モジュールに電源を供給する 5V 電源ピンと GND（接地）ピンに対応している．アナログセンサーのピン 1 はプライマリアナログ入力ピン，ピン 2 はセカンダリアナログ入力ピンという機能をそれぞれ持つ．また，デジタルセンサーのピン 1 はプライマリデジタル入出力ピン，ピン 2 はセカンダリデジタル入出力ピンという機能をそれぞれ持つ．ここでの入力とは，センサーが取得した情報を接続先に出力するという意味であり，出力とは，センサーに対し情報を入力するという意味で用いられている．したがって，アナログセンサーは全て，アナログ値を接続先に対し出力する機能のみ持ち，デジタルセンサーは全て，デジタル値を接続先に出力する機能とデジタル値を接続先から受け取る機能の両方を持つ．

3.2.2 GrovePi+の仕様

ここでは，GrovePi+の仕様について説明する．GrovePi+の主な仕様 [6] をまとめたものを，表 3.1 に示す．表 3.1 にあるように，GrovePi+に搭載されている Grove コネクタの個数は 15 個であるが，それぞれのコネクタで接続できる Grove モジュールのタイプが予め指定されている．GrovePi+に搭載されている Grove コネクタの配置 [7] については，図 3.1 の GrovePi+の模式図に示した．図 3.1 は，GrovePi+を上部から見た様子を模式的に示したものである．4 つの隣接した四角形を内包する四角形 1 つが Grove コネクタを表し，その 4 つ並んだ四角形 1 つが Grove コネクタのピンを表している．Grove コネクタ全てに振られた大文字アルファベット 1 文字が，そのコネクタに接続可能なモジュールのタイプを表している．A がアナログセンサー，D がデジタルセンサー，S が GroveUART モジュール，I が GroveI2C モジュールをそれぞれ意味する．また，ピンを表す四角形に振られた記号について説明する．G は接地ピン（ピン 4），V は 5V 電源ピン（ピン 3），数字は GrovePi+がピンを特定する際

表 3.1: GrovePi+の主な仕様

大きさ	88mm × 58mm × 23mm
重量	51g
電源ソース	Raspberry Pi の周辺機器接続端子
Grove コネクタ	15 個
デジタルセンサー接続用コネクタ	7 個
アナログセンサー接続用コネクタ	3 個

に用いる番号と対応した整数値，CL は I^2C で用いられる SCL と接続されたピン，DA は I^2C で用いられる SDA と接続されたピンとなっている．

次に，本研究で用いるアナログセンサーとデジタルセンサーを接続する，アナログセンサーに対応する Grove コネクタとデジタルセンサーに対応する Grove コネクタの仕様について説明する．ここで，前者をアナログポート，後者をデジタルポートと呼ぶこととする．アナログポートは，接地ピン，電源ピンの他に，14 から 17 という番号のついたピンで構成される．14 番から 17 番のピン全てにアナログ/デジタルコンバータ（A/D コンバータ）が使用されている．この A/D コンバータは，アナログ値をデジタル値に変換する機能を持ち，その解像度は 10 ビットである．そのため，アナログポートでは 10 進数で 0 から 1023 までの整数値を取得することができる．デジタルポートは，接地ピン，電源ピンの他に，2 から 9 という番号のついたピンが使用されている．そのうち，3 番，5 番，6 番，9 番ピンには，PWM（Pulse Width Modulation：パルス幅変調）がサポートされている．PWM をサポートするピンでは，8 ビットの値を接続されたセンサーに対し，出力することができる．それ以外のピンは，0 か 1 のデジタル値のみセンサーに対して出力することができる．

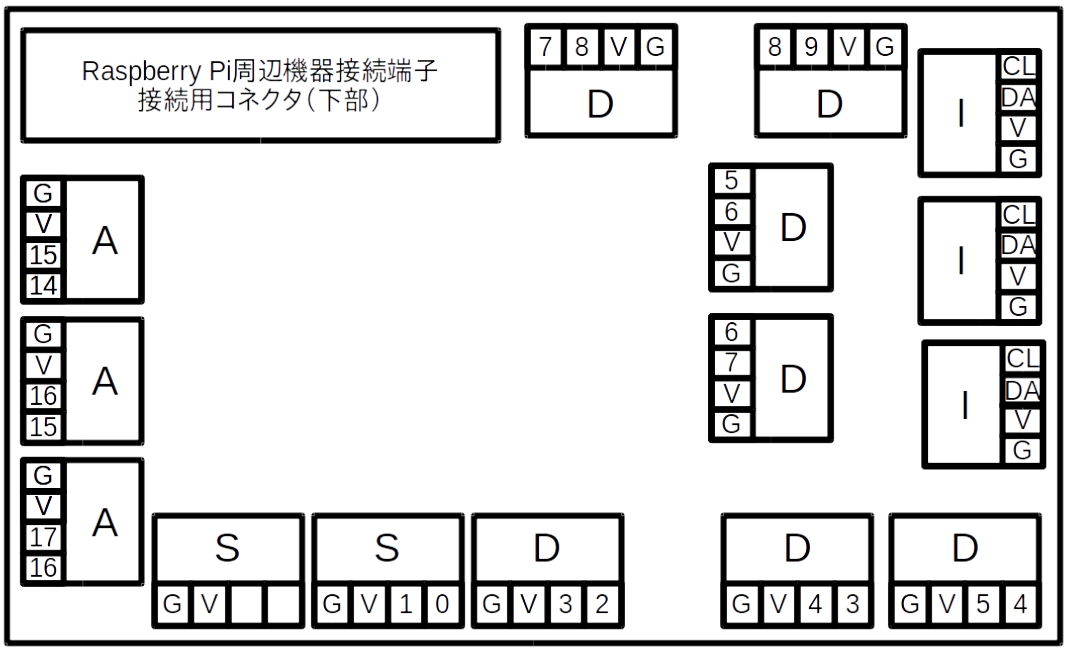
模式図のなかで，ピンに割り振られた記号が他のピンとかぶる箇所が複数存在する．これは，GrovePi+の基板上で物理的に接続されていることを意味する．この影響については，3.3.2 で述べる．

次に，Raspberry Pi と GrovePi+の通信方式について説明する．Raspberry Pi と GrovePi+は I^2C を用いてデータのやり取りを行う．その際，マスターは Raspberry Pi でスレーブは GrovePi+である．GrovePi+に割り当てられている I^2C で用いられる 7 ビットのアドレスは，16 進数にして 0x04 である．したがって，GrovePi+はすべて Raspberry Pi の要求でのみ動作する．また，GrovePi+上の GroveI2C モジュールに対応した Grove コネクタにあるピンには，SDA と SCL に対応したものがある．これは，Raspberry Pi の SDA，SCL とそれぞれ GrovePi+の基板上で物理的に接続されている．

GrovePi+に接続する Grove モジュールは，Raspberry Pi から直接操作することはできない．そのため，Raspberry Pi は GrovePi+を I^2C を用いて操作し，間接的に Grove モジュールを操作することになる．Grove モジュールを操作する方法については，3.3 で詳細を述べる．

3.3 Raspberry Pi を用いた GrovePi+の操作方法

ここでは，Raspberry Pi に GrovePi+を接続し，さらにその GrovePi+に Grove モジュールを接続し，Raspberry Pi から GrovePi+を介して Grove モジュールを操作する方法について詳細に述べる．Raspberry Pi から GrovePi+を操作するためには，Raspbian に多くのライブラリやパッケージをインストールする必要がある．3.3.1 では，Raspberry Pi から GrovePi+を操作できる動作環境の構築方法について説明する．



上部から見たGrovePi+の模式図

図 3.1: GrovePi+の基板の模式図

Raspberry Pi と GrovePi+とのやり取りは，全て I^2C を用いた通信によって行われる．GrovePi+は Raspberry Pi との通信におけるプロトコルを定めており，そのプロトコルに則って I^2C を用いた通信を行わなければいけない．3.3.2 では，そのプロトコルについて説明した後，Raspberry Pi と GrovePi+との間でやり取りされる具体的なデータについて述べる．

3.3.1 Raspberry Pi のセットアップ

Raspberry Pi から GrovePi+を操作するための動作環境の構築は，GrovePi+ Starter Kit に付属していたユーザーマニュアルのセットアップのページに書いてあるとおりに行えば良い．ユーザーマニュアルに書いてある方法は以下の通りである．

1. Raspberry Pi で以下のコマンドを実行し，Dexter Industries が配布しているセットアップツールがあるレポジトリをクローンする．`git clone https://github.com/DexterInd/GrovePi.git`
2. `GrovePi` というディレクトリをクローンできたら，`GrovePi/Script/`へ移動する．
3. `install.sh` というスクリプトファイルを実行ファイルにし，root 権限で実行する．
4. 画面に Reboot という表示がなされたら，Raspberry Pi を再起動する．

この方法を実行すると，自動で Raspberry Pi に GrovePi+を操作できる動作環境が構築される．

これでは，実際に Raspberry Pi に対しどのような処理を加えたのかわからないため，セットアップ方法にある `install.sh` ファイルを調べた．以下に `install.sh` ファイルが行う処理を簡単にまとめる．

1. Raspberry Pi がインターネットに接続しているかどうかを検査
2. `install.sh` を root 権限で実行しているかどうかを検査
3. 必要なライブラリやパッケージのインストール
4. WiringPi というライブラリのビルド及びインストール
5. Raspberry Pi のデバイスに関するブラックリストから，I^2C と SPI を除外
6. I^2C と SPI に関するデバイスファイルを`/etc/modules/`に追加
7. `vrdude_5.10-4_armhf.deb` をインストールし，適切な設定を行う
8. Raspberry Pi の再起動を促す

ここで，SPI とはシリアル通信の規格の 1 つで，Raspberry Pi から GrovePi+のファームウェアを書き換える際に，SPI を用いた通信を行う．本研究では，ファームウェアを書き換えることは行わないため，SPI についての説明は省略する．また，インストールが必要なライブラリやパッケージは以下に示す．

- python-pip
- git
- libi2c-dev
- python-serial
- python-rpi.gpio
- i2c-tools
- python-smbus
- python3-smbus
- arduino
- minicom

python に関連するライブラリが多い理由は，Dexter Industries が提供する Raspberry Pi から GrovePi+を操作するサンプルプログラムの多くが python で記述されていることや，上で述べた `install.sh` に書かれた一部の処理は python のプログラムで記述されているためである．

本研究では，Raspberry Pi をインターネットに接続しないため，一部の作業を手作業で行った．具体的には，まず初めにセットアップツールのクローンを Raspberry Pi とは別の Linux が導入されたコンピュータで行い，得られたディレクトリを全て `scp` コマンドを用いて Raspberry Pi に転送した．その後，必要なライブラリやパッケージを全て Debian のパッケージ管理サイト [8] と python の公式パッ

ケージ配布サイト [9] からクローンした際に用いたコンピュータでダウンロードし，scp コマンドを使用して全て Raspberry Pi に転送し，dpkg -i コマンドを用いて全てインストールを行った．Debian のパッケージ管理サイトからダウンロードする際のアーキテクチャは，Raspberry Pi のアーキテクチャである armhf を選択した．インストールを実行後，install.sh において，パッケージやライブラリをインストールするまでのコマンドの部分を，全てコメントアウトし実行した．以上で，Raspberry Pi から GrovePi+を操作するための動作環境を整えることができた．

3.3.2 GrovePi+の通信プロトコルを用いた操作方法

ここでは，Raspberry Pi に接続する GrovePi+とその GrovePi+に接続された Grove モジュールの操作方法について説明する．Raspberry Pi は，GrovePi+に接続する Grove モジュールを直接操作することはできないため，I^2C を用いて GrovePi+と通信を行い，間接的に Grove モジュールを操作する．まずは，Raspberry Pi と GrovePi+の I^2C を用いた通信で行うやり取りの，大まかな流れについて以下に示す．

1. マスターである Raspberry Pi が I^2C を用いて通信を開始する
2. Raspberry Pi は 5 バイトのデータを GrovePi+に送信する
3. GrovePi+は 5 バイト分のデータを受信すると，そのデータにしたがって GrovePi+に接続されている Grove モジュールを操作する
4. Raspberry Pi は 5 バイトのデータを送信後，必要なら一定時間待機する
5. Raspberry Pi は待機後，必要なら GrovePi+に対しデータの送信を要求する

このように，Raspberry Pi が GrovePi+に対し Grove モジュールの操作を要求するには，GrovePi+のプロトコル [10] に沿って 5 バイトのデータを送信することで行われる．GrovePi+は，Raspberry Pi から送信された操作の内容が Grove モジュールからデータを取得することであった場合，GrovePi+は Grove モジュールからデータを取得し，そのデータを処理した上で GrovePi+上の特定のレジスタに順に格納する．Raspberry Pi から GrovePi+に対し I^2C を用いてデータの送信を要求した場合，この時点で GrovePi+はそのレジスタに格納されている値を順に送信する．Grove モジュールごとで GrovePi+がそのモジュールに対し処理を行うために必要な時間が違うため，そのモジュールに応じた時間だけ GrovePi+が処理を完了するまで Raspberry Pi は待機しなければいけない．また，情報を取得することが可能な Grove モジュールを操作する場合，そのモジュールごとで取得するデータのサイズが違い，それに応じて GrovePi+が処理した後のデータサイズもモジュールごとで変化する．そのため，Raspberry Pi が GrovePi+からデータを受信するときは，そのモジュールごとで適切なサイズのデータを受信する必要がある．

次に，GrovePi+の通信プロトコルに沿って Raspberry Pi が GrovePi+に対し送信する 5 バイトのデータ構造について説明する．I^2C を用いて通信を行うため，8 ビットつまり 1 バイトずつデータは送信される．Raspberry Pi が GrovePi+に送信する 5 バイトのデータのうち，最初に送られる 1 バイトの値は必ず定数 1 でなければいけない．次に送信する値は，GrovePi+が Grove モジュールに対して行う処理に対応する値である．GrovePi+は，Grove モジュールそれぞれに対して行う処理を個別に用意しており，その処理命令ごとに異なる非負整数の番号をつけている．2 バイト目には，その番号の値を送信する．3 バイト目は，GrovePi+上のピンの番号を送信する．このピンの番号とは，図 3.1 中のピン

に割り当てられた非負整数の値のことであり，ここで送信した数値に対応するピンに対し，GrovePi+は指定された処理を行う．4 バイト目と 5 バイト目には，2 バイト目で送信した命令の引数となる値を送信する．

Raspberry Pi から GrovePi+に対し送受信を要求する際は，必ずはじめに 1 バイトの値である 1 を送信する必要がある．Raspberry Pi が GrovePi+に対し処理を要求する際に送る 5 バイトのデータの 1 バイト目が定数 1 であるのは，このためである．Raspberry Pi が GrovePi+からデータを受信する場合も，Rasberry Pi は初めに定数 1 を送信した後で，受信を要求しなければいけない．その際，I^2C では送信と受信の切り替えを行う必要がある．ここでは，ストップコンディションを送信した後再度スタートコンディションを送信することで送受信の変更を行うのではなく，リピートスタートコンディションを用いて送受信の変更を行う必要がある．

GrovePi+に複数の Grove モジュールが接続されている場合，それらを全て Rasoberry Pi から操作する方法について説明する．GrovePi+のプロトコルにあるように，GrovePi+に対して要求できる処理は，1 度に 1 つの処理である．したがって，Raspberry Pi から複数の Grove モジュールを操作する場合でも，1 つずつ Grove モジュールを操作しなければいけない．また，GrovePi+のレジスタ内に情報を格納する処理を連続して要求する場合は，先に格納した情報に上書きするように GrovePi+は情報を格納する．そのため，GrovePi+内のレジスタに情報を格納する処理を行った後は，毎回 Raspberry Pi から GrovePi+に対して情報の送信を要求しなければいけない．なお，GrovePi+内のレジスタの情報は，GrovePi+が次の情報の格納処理を行うまで保持される．

図 3.1 にあるように，GrovePi+上の Grove コネクタには，同じ番号のついたピンが複数ある．Raspberry Pi が 5 バイトのデータを送信することで指定する命令は，指定したピンの番号がつく箇所全てに対して実行される．したがって，GrovePi+に複数の Grove モジュールを接続する際は，ピンの配置を意識して接続する必要がある．なお，GrovePi+のピンがかぶっている箇所は全て，あるコネクタのピン 1 とそれ以外のコネクタのピン 2，という組み合わせで配置されている．多くの Grove センサーはセカンダリピンを使用しない作りとなっているため，複数の Grove センサーを接続しても，ピンの番号がかぶっている部分の他方は Grove センサーの使用されないピンと接続される．そのため，Grove センサーを複数使用する場合は，指定した命令を対応しないセンサーに対し実行されることはほぼ無い．

第4章　IoTデバイスのハードウェア層の実装

4.1　IoTデバイスの概要

本章と5章で，実際にIoTを実装する方法について述べる．ここではまず，実装するIoTの概要について述べる．IoTデバイスは，ハードウェアを制御する仕組みを持つハードウェア層と，IoTデバイスが持つ情報を高水準言語に見せるための仕組みを持つプレゼンテーション層に分けて，それぞれ実装を行う．ハードウェア層は，Raspberry PiとGrovePi+並びに複数のGroveセンサーを用いて実装する．Raspbery Piはハードウェア層全体の制御を担い，GrovePi+とGroveセンサーはIoTデバイス周囲の環境情報を取得する機能を担う．プレゼンテーション層は，HTTPという通信プロトコルを用いた通信を導入し，HTTP通信を用いてIoTデバイスが持つ情報を高水準言語に提供する．したがって，プレゼンテーション層は，HTTPにしたがって情報の提供を行うサーバー側と情報の取得を行うクライアント側に分けられる．

本章では，IoTデバイスのハードウェア層について仕様と実装方法について述べ，5章では，IoTデバイスのプレゼンテーション層の仕様と実装方法について述べる．

4.2　ローカルネットワークの構築

初めにIoTデバイスを接続するローカルネットワークを構築する．安全のため，このネットワークとインターネットは接続しないこととする．IoTデバイスのプレゼンテーション層は，このネットワークを介してIoTデバイスが持つ情報を伝達する．IoTデバイスのネットワーク制御は，Raspberry Piが行う．

このネットワークは，無線LANブロードバンドルーターであるBUFFALO WZR-AMPG300NH[11]を用いて構築した．このルーターの主な仕様は表4.1に示す．このルーターの無線LAN設定は，東北

表 4.1: BUFFALO WZR-AMPG300NHの主な仕様

無線LAN規格	IEEE 802.11 n/a/g/b
無線LANデータ転送速度	最大54Mbps（IEEE802.11g）
無線LAN周波数範囲	2.4GHz（IEEE802.11g)
有線LAN規格	100BASE-TX
有線LAN対応プロトコル	TCP/IP
有線LANポート数	4個（スイッチングHub）
電源	100V 50/60Hz

大学電気通信研究所が定める無線LANアクセスポイント設置ポリシーに基づき行った．以下の表4.2に，実際の無線LAN設定を示す．ここで通信方式は，混信を防ぐためIEEE802.11aが推奨されてい

表 4.2: BUFFALO WZR-AMPG300NH の無線 LAN 設定

暗号化方式	WPA2 パーソナル（WPA2-PSK）
暗号化アルゴリズム	AES
通信方式	IEEE802.11g
電波強度	最弱

るが，Raspberry Pi の無線 LAN の通信方式が IEEE802.11a に対応していないため，IEEE802.11g を選択した．

次に，IoT デバイスのハードウェア層を構成する Raspberry Pi をこのネットワークに接続する．Raspberry Pi のネットワーク接続方法については，2.3.2 で述べたとおりである．本研究では，Raspberry Pi の有線 LAN と無線 LAN の両方を，このネットワークに接続した．Raspberry Pi がネットワークに接続すると，ルーターの DHCP により有線 LAN と無線 LAN のどちらに対しても自動で IP アドレスが割り当てられる．しかし，IoT デバイスの IP アドレスがルーターの DHCP によってその時々で変更されるのは都合が悪いため，Raspberry Pi 側で有線 LAN と無線 LAN それぞれの IP アドレスを固定した．Raspberry Pi の IP アドレスを固定する方法は，2.3.2 で述べたとおりである．この作業は，このネットワークに新たにコンピュータを接続し，そのコンピュータから Raspberry Pi に SSH 接続でログインすることで行った．ここで用いたコンピュータの OS は Windows であり，Windows 上には Virtual Box という仮想マシンが構成されている．この仮想マシンには，Linux の Ubuntu16.04 がインストールされており，この Ubuntu16.04 を用いて Raspberry Pi の操作を行った．これ以降，本論文で IoT デバイスを実装する際に行う Raspberry Pi の操作は全て，この Ubuntu16.04 から SSH 接続により遠隔で行うものとする．この Windows コンピュータは，無線 LAN を用いてこのネットワークに接続している．Windows コンピュータの IP アドレスは，ルーターの DHCP による IP アドレスの固定機能を用いて，ルーター側で固定した．ここまでの設定で，本ネットワークに接続されているデバイスとそのデバイスの IP アドレスの対応表を表 4.3 に示す．

表 4.3: ローカルネットワークに接続するデバイスの IP アドレス

WZR-AMPG300NH	192.168.11.1
Windows PC	192.168.11.20
Raspberry Pi（有線 LAN）	192.168.11.21
Raspberry Pi（無線 LAN）	192.168.11.23

4.3 Raspberry Pi と GrovePi+を用いたハードウェア層の実装

4.3.1 ハードウェア層の仕様

ここでは，IoT デバイスのハードウェア層の仕様を述べる．ハードウェア層は，Raspberry Pi と GrovePi+並びに Grove センサーで構成され，全体の制御は Raspberry Pi が行う．この層の役割は，Grove センサーから取得できる情報を全て取得し，それらの情報について適切な処理を行って，プレゼンテーション層へ渡すことである．この流れを以下に示す．

1. Raspberry Pi が GrovePi+と通信を開始
2. Raspberry Pi が GrovePi+に対し，Grove センサーの情報の取得を行うよう要求する．
3. GrovePi+は要求にしたがって，Grove センサーから情報を取得し，GrovePi+ のレジスタ内に情報を格納する．
4. Raspberry Pi は，GrovePi+が取得したセンサーの情報を受信する．
5. Raspberry Pi は，受信したデータを適切に処理し，プレゼンテーション層に渡す．

また，Raspberry Pi はネットワークの制御も行う．この IoT デバイスは，4.2 で構築したネットワークに常に接続しているものとする．そこで，Raspberry Pi の有線 LAN を用いて，IoT デバイスは常時ネットワークに接続している状態とした．そのため，この IoT デバイスの IP アドレスは，Raspberry Pi の有線 LAN に割り当てられた 192.168.11.21 となる．Raspberry Pi の無線 LAN は，ネットワークを構成するルーターに接続可能なものの，数時間経つと接続ができない状態になることが頻繁にあった．Raspberry Pi とルーターの相性が良くないと推測されるが，実際の原因についてはわからないため，無線 LAN は使わない方針とした．

本研究では，IoT デバイス周辺の気温，湿度，光量，IoT デバイスから障害物までの距離の 4 つの情報を，IoT デバイスが持つ情報とする．これを実現するために用いた 3 つの Grove センサーを，以下に示す．なお，用いた Grove センサーは全て，GrovePi+ Starter Kit for Raspberry Pi に付属するものである．

- Grove - Ultrasonic Ranger Sensor
- Grove - Temperature Humidity Sensor
- Grove - Light Sensor

以上 3 つのセンサーを全て，Grove ケーブルを用いて GrovePi+の適切な Grove コネクタに接続する．これらの詳細な仕様と，GrovePi+のどのコネクタにどのセンサーを接続するかについては，4.3.2 で述べる．

4.3.2 ハードウェア層で用いる各種 Grove センサーの仕様

ここでは，ハードウェア層で用いる Grove センサー各種の仕様についてそれぞれ述べる．3.1 で述べたように，Grove コネクタを持つ Grove モジュールのなかで，周囲の環境情報を取得できるセンサーが搭載されたものを，Grove センサーと呼ぶこととする．以下に，ハードウェア層で用いた 3 種類の Grove センサーについて，その詳細をまとめる．

- Grove - Ultrasonic Ranger Sensor[12]
  このセンサーは，センサー部分から障害物までの距離を超音波を用いて測定するデジタルセンサーである．GrovePi+のデジタルポートに接続し，GrovePi+が行う処理によって，距離を測定し，測定したデータを GrovePi+に渡す．GrovePi+は，このセンサーから情報を受け取ることしかしないため，このセンサーは GrovePi+のデジタルポート全てに接続可能である．このセンサーの主な仕様を，表 4.4 に示す．表 4.4 に示すように，このセンサーで測定できるデータは，全て単位がセンチメートルの整数値である．

表 4.4: Grove - Ultrasonic Ranger Sensor の主な仕様

大きさ	50mm × 25mm × 16mm
重さ	16g
電源ソース	GrovePi+の電源ピン
使用する超音波の周波数	40kHz
測定可能な距離	3cm から 350cm

次に，このセンサーを操作するために必要な情報について説明する．このセンサーは，Grove コネクタのピン 2 は使用しないため，ピン 1 が接続された GrovePi+上のピンの番号を指定する．また GrovePi+が，このセンサーを操作するために用意している処理命令の番号は 7 である．GrovePi+は，このセンサーが接続されたピンに対して 7 番の処理を実行すると，センサーから障害物までの距離が得られる．この距離の値は，GrovePi+によって 16 ビットの値に処理されて GrovePi+ のレジスタ内に格納される．この 16 ビットのデータは，センサーから対象物までの距離をセンチメートルで表した整数値である．GrovePi+が 7 番の処理を実行してから，距離データ 16 ビットの格納が完了するまで，約 50 ミリ秒必要である．Raspberry Pi が GrovePi+に対して，このセンサーの操作を要求した後，50 ミリ秒以上待機し，Raspberry Pi が GrovePi+に対して情報の送信を要求したとき，通信は I^2C 通信で行われるため 8 ビットずつデータの送信が行われる．GrovePi+はまず初めの 8 ビットは用途不明の値を送信し，次から距離データの上位 8 ビット，下位 8 ビットの順で送信を行う．したがって，Raspberry Pi はこのセンサーの情報を得るために，3 バイトの情報の送信を GrovePi+に対して要求する必要がある．なお，このセンサーを処理する GrovePi+の 7 番の命令の 2 つの引数には，両方 0 を指定する．

- Grove - Temperature Humidity Sensor[13]
  このセンサーは，センサー周囲の気温と湿度を同時に測定するデジタルセンサーである．GrovePi+のデジタルポートに接続し，GrovePi+が行う処理によって，温度と湿度を同時に測定し，測定したデータを全て GrovePi+に渡す．GrovePi+は，このセンサーから情報を受け取ることしかしないため，このセンサーは GrovePi+のデジタルポート全てに接続可能である．このセンサーの主な仕様を，表 4.5 に示す．

表 4.5: Grove - Temperature Humdity Sensor の主な仕様

大きさ	40mm × 20mm × 8mm
重さ	8g
電源ソース	GrovePi+の電源ピン
測定可能な湿度	20%から 90%
測定可能な温度	0 ℃から 50 ℃

次に，このセンサーを操作するために必要な情報について説明する．このセンサーは，Grove コネクタのピン 2 は使用しないため，ピン 1 が接続された GrovePi+上のピンの番号を指定する．また GrovePi+が，このセンサーを操作するために用意している処理命令の番号は 40 である．GrovePi+は，このセンサーが接続されたピンに対して 40 番の処理を実行すると，センサー周辺

の湿度と温度を取得できる．これら測定値は，どちらも GrovePi+によって 32 ビットの値に処理されて GrovePi+ のレジスタ内に連続して格納される．この 2 つの 32 ビットデータは，温度と湿度の値を単精度浮動小数型で表したものである．GrovePi+が 40 番の処理を実行してから，32 ビットデータ 2 つの格納が完了するまで，約 0.6 秒必要である．Raspberry Pi が GrovePi+に対して，このセンサーの操作を要求した後，約 0.6 秒待機し，Raspberry Pi が GrovePi+に対して情報の送信を要求したとき，GrovePi+はまず初めに用途不明な 8 ビット値を送信し，次に温度データ 32 ビットを上位から 8 ビットずつ順に 4 回に分けて送信を行う．温度データ 32 ビットの送信の次に，湿度データ 32 ビットを温度データを送信した時と同様に送信する．したがって，Raspberry Pi はこのセンサーの情報を得るために，合計で 9 バイトの情報の送信を GrovePi+に対して要求する必要がある．なお，このセンサーを処理する GrovePi+の 40 番の命令の 2 つの引数うち，1 つ目（Raspberry Pi から GrovePi+に対して処理を要求する際に Raspberry Pi が送信する 5 バイトのデータのうちの 4 バイト目）にはこのセンサーに搭載されたセンサー回路の種類によって異なる値が入る．本研究で用いたセンサー回路に対応する値は 0 である．

- Grove - Light Sensor[14]
  このセンサーは，センサー部分の明度を測定するアナログセンサーである．GrovePi+のアナログポートに接続し，GrovePi+が行う処理によって，明度を測定し，測定したデータを GrovePi+に渡す．GrovePi+は，A/D コンバータを使って受け取った値をデジタル値に変換する．このセンサーの主な仕様を，表 4.6 に示す．

表 4.6: Grove - Light Sensor の主な仕様

大きさ	20mm × 20mm
重さ	4g
電源ソース	GrovePi+の電源ピン

次に，このセンサーを操作するために必要な情報について説明する．このセンサーは，Grove コネクタのピン 2 は使用しないため，ピン 1 が接続された GrovePi+上のピンの番号を指定する．また GrovePi+が，このセンサーを操作するために用意している処理命令の番号は 3 である．この 3 番の処理命令は，アナログセンサーから値を取得し，A/D コンバータを用いてデータをデジタル値に変換し，そのデジタル値をレジスタに格納する，という一連の流れを行うものであり，アナログセンサー全般で用いられる命令である．GrovePi+は，このセンサーが接続されたピンに対して 3 番の処理を実行すると，センサー周辺の明度を 10 進数表記の 0 から 1023 の範囲で取得できる．この値は，GrovePi+によって 16 ビットの値に処理されて GrovePi+ のレジスタ内に格納される．この 16 ビットのデータのうち下位 10 ビットが，アナログデータをデジタルデータに変換したものである．GrovePi+が 3 番の処理を実行してから，距離データ 16 ビットの格納が完了するまで，約 30 ミリ秒必要である．Raspberry Pi が GrovePi+に対して，このセンサーの操作を要求した後，30 ミリ秒以上待機し，Raspberry Pi が GrovePi+に対して情報の送信を要求したとき，GrovePi+はまず初めに 8 ビットの用途不明な値を送信し，次からデータの上位 8 ビット，下位 8 ビットの順で送信を行う．したがって，Raspberry Pi はこのセンサーの情報を得るために，3 バイトの情報の送信を GrovePi+に対して要求する必要がある．なお，このセンサーを処理する GrovePi+の 3 番の命令の 2 つの引数には，両方 0 を指定する．

以上 3 つのセンサーの仕様を考慮し，このハードウェア層の GrovePi+のどのピンにどのセンサー

を接続するかを確定する．Ultrasonic Ranger Sensor は，センサーのピン 1 を GrpvePi+上の 3 番ピンに接続する．Temperature Humidity Sensor は，センサーのピン 1 を GrovePi+上の 4 番ピンに接続する．Light Sensor は，センサーのピン 1 を GrovePi+上の 14 番ピンに接続することとした．

4.3.3 ハードウェア層を実現するプログラムの実装

ここでは，Raspberry Pi から GrovePi+を介して間接的に Grove センサーを操作するためのプログラムの実装を行う．実装には，I^2C 通信での操作を必要とするため，その操作が比較的容易に行えるよう C 言語を用いて行った．また，Raspberry Pi で C 言語を用いて I^2C による通信を行う方法については，2.4.2 で述べたとおりである．

ここで実装するプログラムは，Raspberry Pi が任意のタイミングで IoT デバイスが持つ情報を全てプレゼンテーション層へ渡す，というものである．つまり，任意のタイミングで Raspberry Pi が Grove センサーを操作し，ハードウェア層を構成する 3 つの Grove センサー全てから情報を取得し，それを適切な形式に変換する処理を行った後プレゼンテーション層へ渡す，といったプログラムを実装する．そこで，各 Grove センサーから情報を得るための関数を C 言語を用いてそれぞれ実装する．Ultrasonic Raner Sensor から情報を取得するための関数を `ultrasonic` とし，Temperature Humidity Sensor から情報を取得するための関数を `dht` とし，Light Sensor から情報を取得するための関数を `light` とする．これらの関数をプレゼンテーション層で実行することで，プレゼンテーション層に IoT デバイスの情報を全て適切な形式で渡すことができる．

これら 3 つの関数は，その関数の処理中で I^2C 通信を行う．その中でも，GrovePi+に対して処理を要求する際に 5 バイト分のデータを送信しなければいけない部分や，GrovePi+に対しデータの送信を要求する際に必ず定数 1 を GrovePi+に対して送信しなければいけない部分は，3 つの関数ともに共通の作業である．そこで，前者を `write_block` 関数，後者を `read_data` 関数として定義し，3 つの関数中では，これら 2 つの補助関数を用いて処理を実現する．以下に，`write_block` 関数と，`read_data` 関数の定義を示す．なお，これら補助関数中の引数 `fd` は，I^2C 通信を行うために開いたデバイスファイルへのファイルディスクリプタである．また，これら補助関数は処理が全て正常に行われた場合は整数値 0 を返し，異常が発生した場合は，標準エラー出力を行い，その時点で-1 を返す．

```
int write_block(int fd, int id, int pin, int d1, int d2){
  unsigned char data_block[5];
  int ret;

  data_block[0] = 0x01;
  data_block[1] = id;
  data_block[2] = pin;
  data_block[3] = d1;
  data_block[4] = d2;

  ret = write(fd, data_block, 5);
  if(ret < 0){
    fprintf(stderr, "error writing to i2c slave\n");
    return -1;
  }

  return 0;
}

int read_data(int fd, unsigned char *data, int size){
```

```
  int ret;
  int reg = 0x01;

  ret = write(fd, &reg, 1);
  if(ret < 0){
    fprintf(stderr, "error writing to i2c slave\n");
    return -1;
  }

  ret = read(fd, data, size);
  if(ret < 0){
    fprintf(stderr, "error reading from i2c slave\n");
    return -1;
  }

  return 0;
}
```

以上で得られた補助関数を用いて，Grove センサーそれぞれを操作し情報を取得するための 3 つの関数を実装する．ここで，実装する 3 つの関数全ては，I^2C による通信を行うためのデバイスファイルへのファイルディスクリプタを引数 `fd` としてとるものとする．そのため，これらの関数は全て，I^2C を操作するデバイスファイルのオープン作業と通信対象となるスレーブの固定を行った後で使用することを想定し実装する．以下で，実装する関数ごとに実装戦略を述べたあと，実際に実装を行ったプログラムのソースコードを示す．

- `ultrasonic` 関数
  この関数は，I^2C を操作するデバイスファイルへのファイルディスクリプタのみを引数にとる．また返り値は，処理が正常に行われた場合は実際に得られた距離に関する情報を整数値を返すものとし，異常が起きた場合は -1 を返すものとする．GrovePi+に対して処理を要求する部分では，命令番号の値である 7 とセンサーが接続されたピンの番号の値である 3 を用いて，GrovePi+に送信する 5 バイトのデータを作成し，補助関数の `write_block` で送信処理を行う．Raspberry Pi が GrovePi+から得られる情報は全て 8 ビットの情報であるため，8 ビットずつに分けられた距離データを 16 ビットの距離データに変換する処理を行う．ここでは，`unsigned char` 型の配列に GrovePi+から得られる 8 ビットの複数のデータを順に格納する．この配列は，数値データとして扱うことができるため，距離データの上位 8 ビットと下位 8 ビットを格納し，上位 8 ビットが格納された配列の値に整数値 256 をかける．その値に下位 8 ビットが格納された配列の値を足す．こうすることで，8 ビットずつに区切られた 16 ビットのデータを整数値に変換することができる．以上を踏まえて，実装した関数 `ultrasonic` を以下に示す．

  ```
  int ultrasonicRead(int fd){
    int ret;
    unsigned char data[3];

    ret = write_block(fd, 7, 3, 0, 0);
    if(ret < 0) return -1;

    usleep(60000);

    ret = read_data(fd, data, 3);
    if(ret < 0) return -1;
  ```

```
  return data[1] * 256 + data[2];
}
```

- dht 関数

  この関数は，引数に I^2C を操作するデバイスファイルへのファイルディスクリプタと，測定した温度と湿度を格納する単精度浮動小数点型の変数へのポインタを引数にとる．また返り値は，処理が正常に行われた場合は整数値 1 を返し，異常が起きた場合は－ 1 を返すものとする．GrovePi+に対して処理を要求する部分では，命令番号の値である 40 とセンサーが接続されたピンの番号の値である 4 を用いて，GrovePi+に送信する 5 バイトのデータを作成し，補助関数の write_block で送信処理を行う．Temperature Humidity Sensor の測定した情報は，GrovePi+によって 32 ビットの単精度浮動小数点型に変換される．32 ビットデータは 8 ビットずつのデータに分けられて送信されるため，まずは 4 つの 8 ビットデータから 1 つの 32 ビットデータを作る．そのあとで，32 ビットのデータを単精度浮動小数点データとして取り出す．8 ビットのデータ 4 つは全て整数型の値として扱い，それぞれの桁に応じて 256 を複数回かけることで，本来の 32 ビットデータを整数値として読み込んだ場合の値が得られる．その 32 ビットデータを単精度浮動小数点数として読みこめば，温度と湿度のデータが得られる．ここでは，単精度浮動小数点型である float 型変数 1 つと，float 型とデータサイズが同じ 32 ビットの int 型変数 1 つで構成される共用体を用いる．C 言語における共用体の各メンバは同じメモリ領域に存在するため，整数値で表された 32 ビットのデータをその共用体の int 型変数に格納し，同じ共用体の float 型変数を参照することで，32 ビット単精度浮動小数点数を取り出す．以上を踏まえて，実装した関数 dht を以下に示す．

```
int dht(int fd, float *d){
  int ret;
  unsigned char data[9];
  union {float f; int i;} temp, hum;

  ret = write_block(fd, 40, 4, 0, 0);
  if(ret < 0) return -1;

  usleep(600000);

  ret = read_data(fd, data, 9);
  if(ret < 0) return -1;

  temp.i = data[1] +
           data[2] * 256 +
           data[3] * 256 * 256 +
           data[4] * 256 * 256 * 256;
  hum.i = data[5] +
          data[6] * 256 +
          data[7] * 256 * 256 +
          data[8] * 256 * 256 * 256;

  d[0] = temp.f;
  d[1] = hum.f;

  return 0;
}
```

- light 関数
  この関数は，I^2C を操作するデバイスファイルへのファイルディスクリプタのみを引数にとる．また返り値は，処理が正常に行われた場合は実際に得られた明度に関する情報を整数値を返すものとし，異常が起きた場合は− 1 を返すものとする．GrovePi+に対して処理を要求する部分では，命令番号の値である 3 とセンサーが接続されたピンの番号の値である 14 を用いて，GrovePi+に送信する 5 バイトのデータを作成し，補助関数の write_block で送信処理を行う．Raspberry Pi が GrovePi+から得られる情報は全て 8 ビットの情報であるため，8 ビットずつに分けられた明度データを 16 ビットの明度データに変換する処理を行う．変換方法については，ultrasonic 関数で用いたものと同じである．以上を踏まえて，実装した関数 light を以下に示す．

```
int light(int fd){
  int ret;
  unsigned char data[3];

  ret = write_block(fd, 3, 14, 0, 0);
  if(ret < 0) return -1;

  usleep(30000);

  ret = read_data(fd, data, 3);
  if(ret < 0) return -1;

  return data[1] * 256 + data[2];
}
```

以上で実装した 3 つの関数を用いることで，ハードウェア層を構成する全てのセンサーから情報を取得することができる．以上を踏まえて，ハードウェア層の制御を行う処理を C 言語を用いて実装し以下に示す．

```
#include <linux/i2c-dev.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int ultrasonic(int);
int dht(int, float*);
int light(int);

int main(){
  int fd;
  int ret;
  float dht_data[2];
  int distance;
  int lightvalue;

  fd = open("/dev/i2c-1", O_RDWR);
  if(fd < 0){
    printf("failed to open i2c port\n");
    return 1;
  }
```

```
  ret = ioctl(fd, I2C_SLAVE, 0x04);
  if(ret < 0){
    printf("unable to get bus access to talk to slave\n");
    return 1;
  }

  lightvalue = light(fd);
  if(lightvalue < 0) return 1;
  ret = dht(fd, dht_data);
  if (ret < 0) return  1;
  distance = ultrasonic(fd);
  if(distance < 0) return 1;

  /*以下プレゼンテーション層に値を渡す処理*/

  close(fd);

  return 0;
}
```

第5章　IoTデバイスのプレゼンテーション層の実装

4章では，IoTデバイスの概要やIoTデバイスのハードウェアを制御する仕組みを持つハードウェア層の実装を行った．本章では，IoTデバイスがもつ情報を高水準言語に提供する仕組みを持つプレゼンテーション層の実装を行う．5.2でプレゼンテーション層の仕様を述べ，5.1ではプレゼンテーション層に導入した通信プロトコルのHTTPについて説明する．5.3と5.4で，実際にプレゼンテーション層の実装を行う．

5.1　HTTPの概要

HTTP（Hypertext Transfer Protocol）[15]とは，サーバーとクライアントとの間でHTMLドキュメントの送受信を行うためのプロトコルである．HTTP通信の流れは，クライアントからサーバーに対しリクエストメッセージというデータを送信し，サーバーは受信したレスポンスメッセージにしたがって処理を行い，サーバーはその結果をレスポンスメッセージというデータとしてクライアントに送信する．以下にリクエストメッセージとレスポンスメッセージのデータ構造について，本研究で用いたものを述べる．

- リクエストメッセージ
  リクエストメッセージは，リクエストライン，ヘッダ部，ボディ部で構成される．以下に，リクエストメッセージを構成するデータについて説明する．

  – リクエストライン
    リクエストラインには，メソッド名，URI，HTTPのバージョンの情報が含まれる．メソッド名とは，クライアントがサーバーに対して要求する処理を表した名前である．URI（Uniform Resource Identifier）とは，リソースを識別するための文字列のことであり，インターネットのサイトのアドレス等を表すURLがURIに含まれる．本研究では，GETメソッドを用いた．GETメソッドは，サーバーから指定したURIのリソースを取得するものである．以下に，HTTPバージョン1.1を用いて，サーバーのルートディレクトリに存在するリソースを取得する場合のリクエストラインを示す．
    `GET / HTTP/1.1`
    このように，リクエストラインはメソッド名，URI，HTTPバージョンを順に半角スペースで区切った文字列のことである．

  – ヘッダ部
    クライアントに関する情報や，リクエストするデータのタイプなどをヘッダという構造を用いて表す．ヘッダの要素には，フィールド名とその内容が含まれる．1行に記述できるヘッダは1つのみで，このヘッダー部には複数のヘッダを記述することができる．ヘッダ部は

省略することも可能である．本研究では，Host:というフィールド名を持つヘッダのみを使用した．このヘッダは，サーバーがあるホストを示している．以下に，Raspberry Pi の 80 番ポートをホストに指定する場合のヘッダ部を示す．

```
Host:  192.168.11.21:80
```

– ボディ部
ここには，サーバーに送信するデータを含む．リクエストラインで GET メソッドを使用する場合はこの部分は使用しないため，説明は省略する．

- レスポンスメッセージ
レスポンスメッセージは，ステータスライン，ヘッダ部，ボディ部で構成される．以下に，レスポンスメッセージを構成するデータ構造について説明する．

  – ステータスライン
  ステータスラインには，HTTP のバージョン，ステータスコード，メッセージが含まれる．ステータスコードは 3 桁の整数で表され，リクエストが正常に処理された場合は 200 となる．200 以外が含まれていた場合は，リクエストの処理が正常に行われなかったことを表している．リクエストコードで 200 以外の値については，ここでは説明しない．メッセージとは，サーバーが自由に定めたメッセージのことである．

  – ヘッダ部
  この部分に記述されるヘッダについては，リクエストメッセージのヘッダ部と同様である．レスポンスメッセージのヘッダ部には，サーバがデータを送信した日付や，ボディ部にあるデータのサイズ等が記述される．

  – ボディ部
  サーバーがクライアントに送信するデータ本体が，この部分に記述される．したがって，クライアントはこの部分のデータを取り出せば良い．

5.2 プレゼンテーション層の仕様

IoT デバイスのプレゼンテーション層は，HTTP 通信を導入し，HTTP 通信を用いて IoT デバイスが持つ情報を高水準言語に提供する．したがって，プレゼンテーション層は，HTTP にしたがって情報の提供を行うサーバー側と情報の取得を行うクライアント側に分けられる．

まずは，サーバー側の仕様について説明する．サーバー側は，クライアントのリクエストに応じて情報を送信する．そのために，Raspberry Pi に Web サーバーを導入しサーバーを側を実装する．Web サーバーとは，HTTP にしたがってクライアントに対し，Web サーバー内に存在する HTML ドキュメントの情報を送信する機能を持つソフトウェアのことである．本研究で導入する Web サーバーは，Linux への導入が容易である Apache2 とする．Apache2 の導入方法については，5.3.1 で述べる．

Apache2 は，CGI を用いてハードウェア層から IoT デバイスの情報を受け取る．ここで，CGI とは Common Gateway Interface の略で，サーバーが実行するプログラムのことである．ここでは，C 言語を用いて作成する CGI プログラムを Apache2 が実行することで，IoT デバイスの情報をその CGI プログラム中で取得し，その後，標準出力で HTML ドキュメントを出力する．Apache2 は，CGI プログラムを実行し，標準出力で出力された HTML ドキュメントをクライアントに送信する．この CGI プログラムは，Raspberry Pi の`/var/www/html/cgi-enabled/`に格納するものとし，Apache2 には，このディレクトリを CGI プログラムを格納する場所として設定する．

プレゼンテーション層は，IoT デバイスが持つ情報全てを JSON 形式で高水準言語に渡すものとする．ここで，JSON 形式とは，今日のインターネットのデータ送信で多く使用されるデータ形式のことである．JSON 形式のデータを認識できる高水準言語が増えていることから，この IoT デバイスのプレゼンテーション層でもデータを JSON 形式で送信することとした．

クライアント側は，サーバー側にある CGI プログラムの実行を HTTP でリクエストし，それに対する HTTP のレスポンスを受け取る．その後，レスポンスを解析し，JSON 形式の IoT デバイスの持つデータを取り出す．取り出した JSON 形式のデータを JSON を認識することができる高水準言語で読み込むことで，高水準言語から IoT デバイスが持つ情報が見えるようになる．

5.3 プレゼンテーション層のサーバー側の実装

5.3.1 Web サーバー Apache2 の導入

プレゼンテーション層のサーバー側の実装の第一段階として，Raspberry Pi に Web サーバーである Apache2 を導入する．Apache2 をインストールしたら，すぐに Raspberry Pi が Web サーバーとして機能する．ここでは，Apache2 をインストール後に，Apache2 が CGI プログラムを実行できるように設定を行う．

以下の手順で，Raspberry Pi に Apache2 をインストールした．

1. Apache2 のパッケージを Ubuntu16.04 を使用して，Debian のパッケージ配布ページ [8] からダウンロードする．
2. `scp` コマンドを用いて，Raspberry Pi に Apache2 のパッケージを転送する．
3. Raspberry Pi 上で，`dpkg -i` コマンドを用いて，Apache2 をインストールする．

以上の作業を終えた後で，Windows コンピュータ上の Web ブラウザを用いて Raspberry Pi にアクセスすると，Apache2 が用意した Web ページが表示される．よって，Raspberry Pi が Apache2 を用いて Web サーバーとし機能するようになった．

次に，Apache2 が CGI プログラムを実行できるように設定を行う．`/etc/apache2/conf-enabled/cgi-enabled.conf` に以下の記述を追加する．

```
<Directory "/var/www/html/cgi-enabled">
  Options +ExecCGI
  AddHandler cgi-script .cgi
</Directory>
```

その後で，Raspberry Pi を再起動することで，`/var/www/html/cgi-enabled/` というディレクトリに格納されている CGI プログラムを，Apache2 はクライアントからのリクエストにより実行することができる．

デバイスファイル等の読み書きを行う CGI プログラムは，root 権限が必要となる．したがって，Apache2 をインストールしたままの初期設定では，root 権限が必要となる CGI プログラムはクライアントからのリクエストによって実行することができない．この問題の解決策として，Apache2 の CGI プログラムの実行ユーザーを調べ，その実行ユーザーのグループに gpio と i2c を追加する．このようにすることで，CGI プログラムの実行ユーザーは，CGI プログラムから GPIO と I^2C に関する制御

を行えるようになる．Apache2 の実行ユーザーは，/etc/apache2/envvars 中で定義されているため，このファイルの中を確認した．その結果，Apache2 の実行ユーザーは www-data という名前であることがわかった．また，以下のコマンドを実行することで，www-data のグループに gpio と i2c を追加した．

```
sudo gpasswd -a www-data gpio
sudo gpasswd -a www-data i2c
```

5.3.2 IoT デバイスの情報を取得し送信する CGI プログラムの作成

ここでは，クライアントのリクエストによって，IoT デバイスが持つ情報を取得し JSON 形式に変換して送信する CGI プログラムを実装する．CGI プログラムは C 言語を用いて実装を行い，その実行ファイルの名前は json.cgi とした．

クライアントに送信される IoT デバイスの情報は JSON 形式を用いる．JSON 形式のデータに変換する情報は，IoT デバイスの IP アドレス，IoT デバイス周辺の温度，湿度，明度，IoT デバイスに付属する距離測定センサーから障害物までの距離である．IP アドレスのキーは"ip"とし，IP アドレスは文字列型とする．それ以外のデータは"data"というキーのオブジェクト中に存在する．温度のキーは"temperature"とし，温度の値は実数型とする．湿度のキーは"humidity"とし，湿度の値は実数型とする．明度のキーは"light"とし，明度の値は整数型とする．距離のキーは"distance"とし，距離の値は整数型とする．

CGI プログラム中で標準出力を用いて HTML ドキュメントを出力することで，その HTML ドキュメントがクライアントに送信される．したがって，IoT デバイスの情報を取得し，その情報を JSON 形式に変換した後，HTML ドキュメントに埋め込んで標準出力にて出力を行う．IoT デバイスの情報の取得には，4 章で実装した関数を使用することで行う．なお，HTML ドキュメントに関する情報については，説明を省略する．以下に実装したプログラムを示す．

```
#include <linux/i2c-dev.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int ultrasonic(int);
int dht(int, float*);
int light(int);

int main(){
  int fd;
  int ret;
  float dht_data[2];
  int distance;
  int lightvalue;

  fd = open("/dev/i2c-1", O_RDWR);
  if(fd < 0){
    printf("failed to open i2c port\n");
    return 1;
  }
  ret = ioctl(fd, I2C_SLAVE, 0x04);
  if(ret < 0){
```

```
    printf("unable to get bus access to talk to slave\n");
    return 1;
  }

  lightvalue = light(fd);
  if(lightvalue < 0) return 1;
  ret = dht(fd, dht_data);
  if (ret < 0) return  1;
  distance = ultrasonic(fd);
  if(distance < 0) return 1;

  printf("Content-type: application/json\n\n");
  printf("[");
  printf("{");
  printf("\"ip\":\"192.168.11.21\",");
  printf("\"data\":");
  printf("{");
  printf("\"temperature\":%f,", dht_data[0]);
  printf("\"humidity\":%f,", dht_data[1]);
  printf("\"light\":%d,", lightvalue);
  printf("\"distance\":%d", distance);
  printf("}");
  printf("}");
  printf("]");

  close(fd);

  return 0;
}
```

このプログラムの実行ファイルを作成し，CGI プログラムを格納するディレクトリに設置した後で，Windows コンピュータの Web ブラウザを用いて `json.cgi` にアクセスした場合，以下のような表示が得られ，プレゼンテーション層のサーバー側の実装が完了したといえる．なお，使用した Web ブラウザは firefox であり，firefox は JSON 形式のデータを表示する際自動で整形する機能をもつ．そのため，以下でも firefox によって整形された JSON データを示す．

```
[
  {
    "ip": "192.168.11.21",
    "data": {
      "temperature": 26,
      "humidity": 14,
      "light": 761,
      "distance": 206
    }
  }
]
```

5.4 プレゼンテーション層のクライアント側の実装

プレゼンテーション層のサーバー側の実装が完了したため，次にクライアント側の実装を行う．クライアント側は，サーバーに対し HTTP のリクエストメッセージで GET メソッドを用いて`/cgi-enabled/json.cgi`

を指定し，IoT デバイスが持つ情報の取得のリクエストを行う．その後，サーバーから送信されるレスポンスデータを受信し，ステータスラインのステータスコードを確認する．ステータスコードが 200 である場合は，レスポンスメッセージのボディ部のみを取り出し，JSON 形式として認識させる．ステータスコードが 200 以外の場合は，レスポンスメッセージ全体を出力する．

以上の処理を行うプログラムを，Ubuntu16.04 上で C 言語と SML#を用いて実装する．C 言語を用いて，サーバーとの接続を行う関数と文字列の送受信を行う関数をそれぞれ作成し，SML#の C 言語との連携機能 [17] を用いて SML#から C 言語で作成した関数を呼び出し，残りの処理を実装する．なお，SML#は JSON 形式のデータをサポート [17] しており，今回実装するクライアントのプログラムでは，SML#を用いて JSON 形式の IoT デバイスの情報を読み込むまでの処理を実装する．5.4.1 では，C 言語による HTTP 通信を行うための準備の処理を行う関数の実装を行う．また 5.4.2 では，C 言語で実装を行った関数を用いてプレゼンテーション層のクライアント側のプログラムの実装を行う．

5.4.1 C 言語を用いた HTTP 通信の準備を行う関数の実装

ここでは，C 言語を用いて HTTP 通信を行うための処理を実装する．HTTP 通信は，TCP のソケット通信 [16] を行うことで実現する．ソケット通信ではまずソケットを生成し，そのソケットとサーバーのソケットとの接続を行う．ソケット通信の接続が完了すると，サーバーとのデータのやり取りを行うことができ，ソケットを切断することで通信を終了することができる．C 言語を用いて，ソケットの生成とソケット通信の接続を行う関数と，データの送信受信それぞれを行う関数を実装する．ソケットの生成と通信を行う関数を `connectServer` とし，データの送信を行う関数を `writeString`，データの受信を行う関数を `readData` とする．関数 `connectServer` は，ソケットの生成が完了しソケット通信の接続が完了した場合は，生成したソケットの識別子を返し，いずれかの処理で失敗した場合は－1 を返す．関数 `writeString` は，ソケットの識別子と送信するデータの文字列を引数にとり，返り値には実際に送信を行ったデータのバイト数とする．ただし送信に失敗した場合は，－1 を返り値とする．関数 `readData` は，ソケットの識別子を引数にとり，返り値には実際に受信を行ったデータのバイト数とする．ただし受信に失敗した場合は，－1 を返り値とする．これらの関数を以下に示す．

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <netdb.h>
#include <netinet/in.h>
#include <sys/param.h>
#include <unistd.h>

int connectServer(){
  int s;
  struct addrinfo hints, *res;
  struct in_addr addr;
  int err;

  memset(&hints, 0, sizeof(hints));
  hints.ai_socktype = SOCK_STREAM;
  hints.ai_family = AF_INET;

  char *serviceType = "http";

  err = getaddrinfo("192.168.11.21", serviceType, &hints, &res);
```

```
  if(err != 0){
    printf("error %d\n", err);
    return -1;
  }

  s = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
  if(s < 0){
    fprintf(stderr, "ソケットの生成に失敗\n");
    return -1;
  }

  if(connect(s, res->ai_addr, res->ai_addrlen) != 0){
    fprintf(stderr, "connect に失敗\n");
    return -1;
  }

  return s;
}

int writeString(int fd, const char* s){
  int ret = write(fd, s, strlen(s));
  return ret;
}

char* readData(int fd){
  char data[1024];

  int ret = read(fd, data, DATA_SIZE);
  return data;
}
```

ここで，ソケットを生成する関数は `int socket(int, int, int)` である．第一引数はプロトコルの種類に対応する整数値，第二引数にはソケット通信のタイプに対応する整数値，第三引数にはプロトコルの種類に対応する整数値をそれぞれとる．この関数の返り値は，生成したソケットの識別子となる．また，ソケット通信の接続を行う関数は，`int connect(int, struct sockaddr*, int)` である．第一引数には生成したソケットの識別子，第二引数には接続先のサーバーを指定する情報等が格納された構造体のポインタ，第三引数には第二引数のサイズをそれぞれとる．ソケット通信の送信を行う関数は `int write(int, char*, int)` である．第一引数にはソケットの識別子，第二引数には送信する文字列が格納されている変数の先頭のポインタ，第三引数は送信する文字列のサイズをそれぞれとり，返り値は実際に送信を行うことができたデータのバイト数である．ソケット通信の受信を行う関数は `read(int, char*, int)` である．第一引数にはソケットの識別子，第二引数には受信するデータを格納する変数の先頭ポインタ，第三引数には受信するデータのサイズをそれぞれとり，返り値は実際に受信を行うことができたデータのバイト数である．

5.4.2 SML#を用いたクライアント側の処理を行うプログラムの実装

C 言語を用いて実装した関数を用いて，SML#でプレゼンテーション層のクライアント側のプログラムを実装する．このプログラムでは，ソケット通信に関する 3 つの C 言語で実装された関数を SML#の C 言語との連携機能を使ってインポートする．その後，ソケット生成とソケット通信の接続を行う．ソケット通信の接続が成功した場合は，HTTP のリクエストメッセージを送信し，IoT デバイスの情報

の送信をサーバー側に要求する．その後，サーバーから送信されるレスポンスメッセージを受信し，そのレスポンスメッセージを解析する．ステータスコードが 200 であれば，レスポンスメッセージのボディ部には JSON 形式の文字列データがあるため，そのデータをすべてとりだす．ステータスコードが 200 以外であった場合は，レスポンスメッセージを全て出力し，プログラムを終了する．取り出した JSON 形式のデータを，SML#の JSON サポート機能を用いて読み込み，SML#の JSON データのプリンタを用いて JSON 形式のデータを標準出力する．

以上の実装戦略のもとで，クライアント側のプログラムを実装する．ソケット通信を用いてデータを受信する関数を `readString` とし，その型は `int -> string` とする．この関数は，ソケットの識別子を引数にとり，ソケット通信で受信した文字列を返す．ソケット通信を用いてデータを送信する関数を `writeString` とし，その型は `(int * string) -> ()` とする．この関数は，ソケットの識別子と送信するデータの文字列の組を引数とし，返り値は無い．`writeString` 関数を用いて HTTP のリクエストメッセージを送信する関数を `sendHTTP` とし，その型は `(string * string * string * string) -> int -> ()` とする．この関数は，サーバーの IP アドレス，接続先のポート番号，サーバーのルートディレクトリから対象のファイルまでのパス，対象のファイル名をそれぞれ文字列で表し，この 4 つの文字列の組とソケットの識別子を引数とし，返り値は無い．`sendHTTP` 関数と `readData` 関数を用いて，サーバーからレスポンスメッセージを受信するまでの処理を行う関数を `getHTML` とする．この関数は，`sendHTTP` 関数の引数である 4 つの文字列データの組を引数にとり，サーバーから受信したレスポンスメッセージ全体の文字列を返り値とする．これら関数を以下に実装する．

```
val connect = _import "connectServer" : () -> int

val r = _import "readData" : int -> char ptr
fun readString s = Pointer.importString (r s)

val w = _import "writeString" : (int, string) -> int
fun writeString (socket, s) =
let
  val ret = w (socket, s)
in
  if ret >= 0 then ()
  else print "write error\n"
end

fun sendHTTP (host, port, path, file) socket = (
  writeString (socket, "GET " ^ path ^ file ^ " HTTP/1.0\r\n");
  writeString (socket, "Host: " ^ host ^ ":" ^ port ^ "\r\n");
  writeString (socket, "\r\n"))

fun getHTML (host, port, path, file) =
let
  val socket = connect ()
  val _ = sendHTTP (host, port, path, file) socket
in
  if socket < 0 then "connect error\n"
  else readString socket
end
```

ここで，`getHTML` に引数 `("192.168.11.21", "80", "/cgi-enabled/", "json.cgi")` を与えてその文字列を標準出力に出力した結果が以下のとおりである．なお，以下の出力では適切な位置で一部改行を加えた．

```
HTTP/1.1 200 OK
Date: Sun, 04 Feb 2018 10:18:02 GMT
Server: Apache
Connection: close
Content-Type: application/json

[{"ip":"192.168.11.21",
"data":{"temperature":26.000000,"humidity":14.000000,"light":760,"distance":207}}]
```

次に，この出力結果からステータスコードの値を確認し，200 であった場合は JSON データのみを文字列で取り出す関数を parseResponce とする．この関数は，文字列のレスポンスメッセージを引数にとる．そのレスポンスメッセージの 10 文字目から 3 文字だけ文字列として取り出す．この 3 文字がステータスコードである．レスポンスメッセージのステータスラインの仕様上，必ず 10 文字目からステータスコードが始まるためこのような処理を選択した．このステータスコードの値によってパターンマッチングを行い，200 であれば JSON 形式のデータを取り出す処理を行い，その JSON 形式のデータを文字列として返す．ステータスコードが 200 以外の場合は，レスポンスメッセージをそのまま返す．また，JSON 形式のデータの取り出しについて説明する．ステータスコードが 200 である場合のレスポンスメッセージのヘッダ部は，必ず 4 行のみである．そのため，ステータスラインとヘッダ部，ヘッダ部とボディ部の間の改行の計 6 行を，レスポンスメッセージの 1 行目から削除し，残った文字列がボディ部の内容である．このような処理を行う関数 parseResponce を以下に示す．

```
fun parseResponce s =
  case substring (s, 9, 3) of
       "200" =>
       let
         val sl = explode s
         fun cut nil _ = ""
           | cut (#"\n" :: t) 5 = implode t
           | cut (#"\n" :: t) n = cut t (n + 1)
           | cut (h :: t) n = cut t n
       in
         cut sl 0
       end
     | _ => s
```

以上で実装した関数を用いて，JSON 形式の IoT デバイスの情報を読み込み，SML#が解釈した IoT デバイスの JSON 形式のデータを出力するプログラムを以下に実装する．

```
val responce = getHTML ("192.168.11.21", "80", "/cgi-enabled/", "json.cgi")
val _ = print ("#responce:\n" ^ responce ^ "\n")

val J = parseResponce responce
val _ = print ("#parse responce:\n" ^ J ^ "\n")

val j = JSON.import J
val _ = print ("#json:\n" ^ JSON.jsonDynToString j ^ "\n")
```

また，以上を実行した際に得られる出力結果を以下に示す．

```
#responce:
```

```
HTTP/1.1 200 OK
Date: Sun, 04 Feb 2018 10:18:02 GMT
Server: Apache
Connection: close
Content-Type: application/json

[{"ip":"192.168.11.21",
"data":{"temperature":26.000000,"humidity":14.000000,"light":760,"distance":207}}]

#parse responce:
[{"ip":"192.168.11.21",
"data":{"temperature":26.000000,"humidity":14.000000,"light":760,"distance":207}}]

#json:
[
 {
    data:{distance:207, humidity:14.0, light:760, temperature:26.0},
    ip:"192.168.11.21"
 }
]
```

この結果から，プレゼンテーション層のクライアントの部分を実装することができ，IoT デバイスの情報を高水準言語から見ることができるようになった.

第6章　現状と今後の課題

6.1　現状のまとめ

本研究では，Raspberry Pi と GrovePi+並びに Grove センサーを組み合わせて，IoT デバイスのハードウェア層の実装を行うことができた．また，Raspberry Pi に Web サーバーである Apache2 を導入し Raspberry Pi と HTTP 通信を行える環境を構築することで，IoT デバイスのプレゼンテーション層の実装を行うことができた．ハードウェア層では，GrovePi+に 3 つの Grove センサーを接続し，その GrovePi+と Raspberry Pi を接続し，Raspberry Pi から GrovePi+を介して Grove センサーを操作するための仕組みを，C 言語を用いて実装した．その仕組みを用いてプレゼンテーション層に Grove センサーの情報を全て渡すことができる．プレゼンテーション層では，サーバー側とクライアント側に分けてそれぞれ実装を行った．サーバー側は，Apache2 が CGI プログラムを実行できるよう設定を行い，ハードウェア層が提供するプログラムを CGI プログラムとして実行することで，クライアント側に HTTP 通信を行うことで IoT デバイスの持つ情報を提供することができた．また，IoT デバイスの持つ情報は JSON 形式で送信され，クライアント側は JSON 形式のデータを認識できる SML#を用いることで，JSON 形式で送信される IoT デバイスの情報を扱うことができた．以上により，IoT デバイスが持つ情報を高水準言語で扱うための仕組みについて実装を行うことができた．

6.2　今後の課題

本研究の最終目標は，IoT デバイスを高水準に扱うための機能を持つオペレーティングシステム（IoT OS）の実装である．1970 年台に開発された VAX システムに導入された VAX/VMS というオペレーティングシステム [1] は，システムの初期化プロシージャーで，接続されたデバイスの初期化を行う．その際，システムは，接続可能な場所全てにおいてデバイスが接続されているかどうかをテストし，存在するデバイスがあれば，そのデバイスの情報をシステムのメモリにマッピングする．このような処理を行うことで，オペレーティングシステムは，そのデバイスの情報を使うことができるようになる．この OS を参考にすると，IoT OS の必要な機能として，ネットワーク内に存在する IoT デバイスの存在を確認する機能，ネットワーク内で存在を確認した IoT デバイスを操作するための準備や初期化を行う機能，初期化を行った IoT デバイスの情報を高水準言語に提供する機能などが挙げられる．また，本研究では IoT デバイス 1 つに対する処理のみを行ったが，IoT OS では複数の IoT デバイスを同時に操作するための仕組みが必要であると考えられる．

本研究では，IoT デバイスの情報等を直接プログラムに埋め込むことで，ハードウェア層とプレゼンテーション層を実装した．そこで，今後の課題として，IoT デバイスを検索しその結果得られる使用可能な複数の IoT デバイスの情報を用いて処理を行うハードウェア層やソフトウェア層の実装が挙げられる．また，それらの情報のサイズ等は IoT デバイスを検索した結果から得られると考えられ，そのデータサイズの情報を用いて IoT デバイスの情報をメモリにマッピングする機能の実装を行い，プレゼンテーション層で行う処理をより OS が行う処理に近づけたいと考えている．

謝辞

本研究を行うにあたり，多大なご指導を賜りました大堀淳教授に深く感謝いたします．本研究を進めるにあたり，数多くのご助言をいただき，熱心な指導をしてくださいました上野雄大准教授に深く感謝いたします．そして，日々の研究室での研究を支えてくださった大堀・上野研究室の皆様に深く感謝いたします．

参考文献

[1] Ruth E. Goldenberg, Lawrence J. Kenah with the assistance of Denise E. Dumas. VAX/VMS Internals and Data Structures VERSION 5.2

[2] 飛松清・奥谷謙一．Go 言語入門 株式会社リックテレコム 2016

[3] Raspberry Pi - Teach, Learn, and Make with Raspberry Pi. URL:http://www.raspberrypi.org

[4] Documentation/i2c/dev-interface - The Linux Kernel Archives. URL:https://www.kernel.org/doc/Documentation/i2c/dev-interface

[5] Grove System - Seeed Wiki - Seeed Studio. URL:http://wiki.seeed.cc/Grove_System/

[6] GrovePi+ - Seeed Studio. URL:https://www.seeedstudio.com/GrovePi%2B-p-2241.html

[7] GrovePi Port description - Dexter Industries. URL:https://www.dexterindustries.com/GrovePi/engineering/port-description/

[8] Debian パッケージ．URL:https://www.debian.org/distrib/packages

[9] PyPI - the Python Package Index. URL:https://pypi.python.org/pypi

[10] GrovePi Protocol and Adding Custom Sensors - Dexter Industries. URL:https://www.dexterindustries.com/GrovePi/programming/grovepi-protocol-adding-custom-sensors/

[11] WZR-AMPG300NH. URL:http://buffalo.jp/products/catalog/network/wzr-ampg300nh/

[12] Grove - Ultrasonic Ranger Sensor - Seeed Studio. URL:https://www.seeedstudio.com/Grove-Ultrasonic-Ranger-p-960.html

[13] Grove - Temperature Humidity Sensor - Seeed Studio. URL:https://www.seeedstudio.com/Grove-Temperature-%26amp%3B-Humidity-Sensor-%EF%BC%88DHT11%EF%BC%89-p-745.html

[14] Grove - Light Sensor -Seeed Studio. URL:https://www.seeedstudio.com/Grove-Light-Sensor-p-746.html

[15] Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC2616(Draft Standard), 1999. URL: https://www.ietf.org/rfc/rfc2616.txt

[16] 小俣光之．C 言語による TCP/IP ネットワークプログラミング．株式会社ピアソンエデュケーション．2001

[17] プログラミング言語SML#解説 http://www.pllab.riec.tohoku.ac.jp/smlsharp/docs/3.4.0/ja/manual.xhtml