ECHONET Lite 機器用通信ミドルウェア TOPPERS/ECNL

一般照明の実装手順書

2015 年 2 月 23 日

長島 宏明

コアーズ株式会社

ECHONET Lite 機器用通信ミドルウェア TOPPERS/ECNL の使用方法として一般照明の実装手順を記載した説明書

ECHONET Lite 機器用通信ミドルウェア TOPPERS/ECNL 一般照明の実装手順書

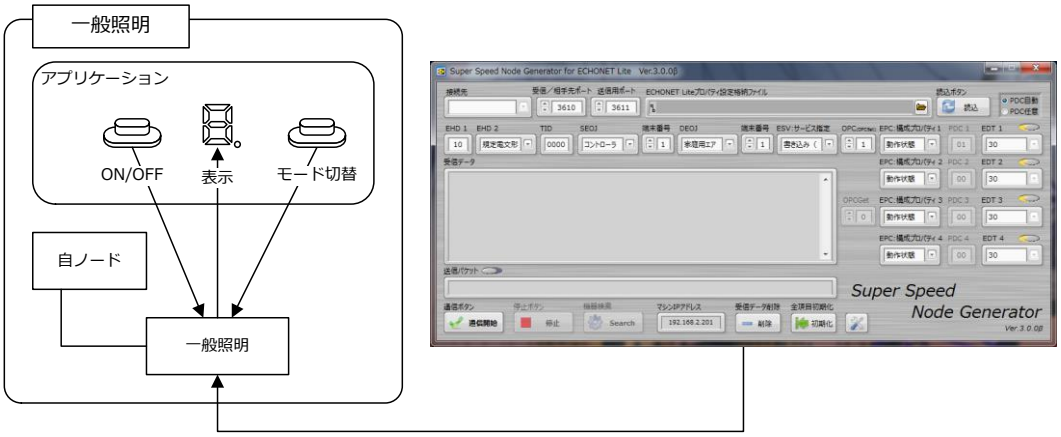
目次

1．一般照明クラスの実装手順

一般照明クラスの実装手順について説明します。

【1】　仕様

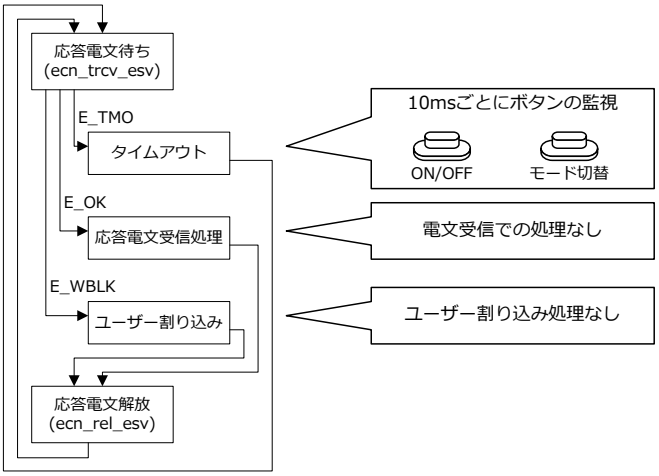
実装する一般照明はボタンを 2 つ持ち、1 つは照明の ON/OFF を行うボタンと、もう 1 つは点灯モードを切り替えるボタンとします。また、7 セグ LED に照明の ON/OFF と、点灯モードを表示します。



ECHONET 機器オブジェクトとして、一般照明クラスを 1 つ持つ自ノードのみの構成とします。

【2】　アプリケーションの動作

アプリケーションの動作を行うタスクは、応答電文を待つループで処理を行います。



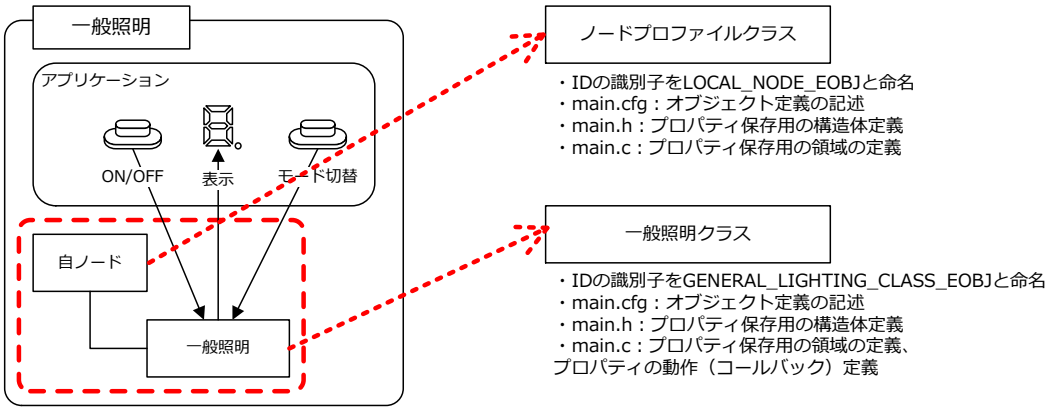
タイムアウト付きの応答電文待ち関数の戻り値で、タイムアウト処理、応答電文受信処理、ユーザー割り込み処理の 3 つに分岐してそれぞれの処理を行います。

今回のアプリケーションでは、タイムアウト処理のみ使用します。タイムアウト処理で定期的にボタンの押されたか否かの監視を行い、状態に変化があった時に所定の処理を行います。

1.2. 機器オブジェクトの定義

一般照明の機器オブジェクト定義の方法を説明します。

【1】　対応するクラス



上図で示した自ノードはノードプロファイルクラス、一般照明は一般照明クラスに対応します。

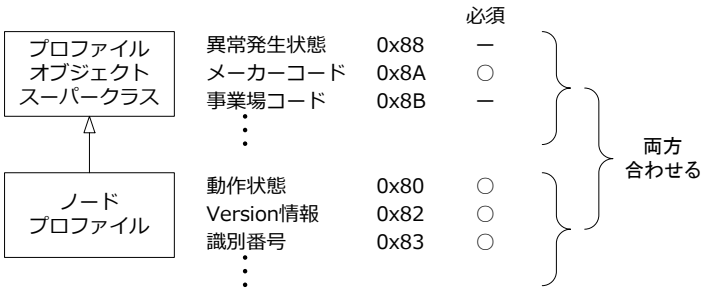
アプリケーションを記述する主なファイルは、main.h、main.c、main.cfg があり、それぞれに以下の内容を記述します。

main.cfg にノードプロファイルクラスと一般照明クラスの定義を静的 API で記述します。

main.h、main.c にプロパティ保存用の構造体定義と、その実体の定義を行います。また、プロパティ設定用のコールバック関数の定義と実装を記述します。

main.h は main.cfg からも参照され、プロパティ保存用の構造体の宣言やプロパティ取得・設定用のコールバック関数の宣言が、静的 API の定義から参照されます。

【2】　自ノード



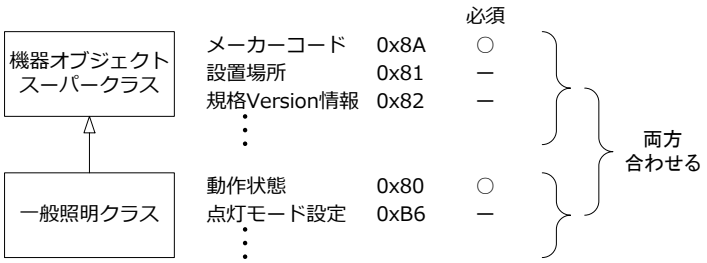
ノードプロファイルクラスは、プロファイルオブジェクトスーパークラスから派生したクラスなので、その両方のプロパティを定義する必要があります。ただし、必須でないものは、必要のある場合のみ定義します。

また、コンフィギュレーターで自動生成するプロパティがありますので、そのプロパティは定義する必要ありません。

自動生成するプロパティは、インスタンス数（0xD3）、クラス数（0xD4）、インスタンスリスト通知（0xD5）、インスタンスリスト S（0xD6）、クラスリスト S（0xD7）、

状態アナウンスプロパティマップ（0x9D)、Set プロパティマップ（0x9E)、Get プロパティマップ（0x9F）です。

【3】　一般照明



一般照明クラスは、機器オブジェクトスーパークラスから派生したクラスなので、その両方のプロパティを見て、実装するものを選択します。機器オブジェクトでも自動生成するプロパティは定義する必要はありません。

自動生成するプロパティは、状態アナウンスプロパティマップ（0x9D)、Set プロパティマップ（0x9E)、Get プロパティマップ（0x9F）です。

【4】　自ノードの cfg ファイル定義

今回選択した自ノードのプロパティを下の表に規格書から転記しました。

プロパティ名称	EPC	プロパティ内容 値域	データ型	データサイズ(Byte)	アクセスルール	必須	状態変化時アナウンス
動作状態	0x80	ノードの動作状態を示す。 0x30=起動中,0x31=未起動中	unsigned char	1	Set Get	○	○
Version情報	0x82	通信ミドルウェアが適用しているECHONET LiteのVersion、および通信ミドルウェアがサポートする電文タイプを示す。 1バイト目：メジャーバージョン（小数点以上）をBinaryで示す。 2バイト目：マイナーバージョン（小数点以上）をBinaryで示す。 3、4バイト目：電文タイプをビットマップで示す。	unsigned char×4	4	Get	○	
識別番号	0x83	オブジェクトを、ドメイン内で一意に識別するための番号。 1バイト目：下位通信層IDフィールド 0x01～0xFD：下位通信層で使用される通信プロトコルで固有の番号が振られている場合、プロトコル種別に応じて、 任意に設定（ECHONET Liteでは使用しない） 0xFE:2～17バイトをメーカ規定形式により設定 0xFF:2～9バイトを乱数により生成するプロトコルを下位通信層で使用する場合に設定 0x00:識別番号未設定 2バイト目以降：固有番号フィールド	unsigned char×9 or unsigned char×17	9 or 17	Get	○	
異常内容	0x89	異常内容 0x0000～0x03E8(0～1000)	unsigned short	2	Get		
メーカーコード	0x8A	3バイトで指定 (ECHONETコンソーシアムで規定。)	unsigned char×3	3	Get	○	

まず、自ノードの ID の識別子を「LOCAL_NODE_EOBJ」とし、静的 API「ECN_CRE_EOBJ」を使用してオブジェクトを定義します。ノードの場合は、第二引数は

「EOBJ_NULL」になります。

次に、自ノードのプロパティを定義していきます。静的 API「ECN_DEF_EPRP」を使用して、規格書の対応する項目を元に定義します。

自ノードのIDの識別子：LOCAL_NODE_EOBJ

```
/*
 *  ノードプロファイルオブジェクト
 */
ECN_CRE_EOBJ (LOCAL_NODE_EOBJ, { EOBJ_LOCAL_NODE, EOBJ_NULL, 0, EOJ_X1_PROFILE, EOJ_X2_NODE_PROFILE,
EOJ_X3_LOCAL_NODE });

/* 動作状態 */
ECN_DEF_EPRP (LOCAL_NODE_EOBJ, { 0x80, EPC_RULE_SET | EPC_RULE_GET, 1,
(intptr_t)&local_node_data.operation_status, (EPRP_SETTER *)onoff_prop_set, (EPRP_GETTER *)ecn_data_prop_get });

/* Ｖｅｒｓｉｏｎ情報 */
ECN_DEF_EPRP (LOCAL_NODE_EOBJ, { 0x82, EPC_RULE_GET, 4, (intptr_t)&local_node_data.version_information,
(EPRP_SETTER *)ecn_data_prop_set, (EPRP_GETTER *)ecn_data_prop_get });

/* 識別番号 */
ECN_DEF_EPRP (LOCAL_NODE_EOBJ, { 0x83, EPC_RULE_GET, 17, (intptr_t)&local_node_data.identification_number,
(EPRP_SETTER *)ecn_data_prop_set, (EPRP_GETTER *)ecn_data_prop_get });

/* 異常内容 */
ECN_DEF_EPRP (LOCAL_NODE_EOBJ, { 0x89, EPC_RULE_GET, 2, (intptr_t)&local_node_data.fault_content, (EPRP_SETTER
*)node_profile_object_fault_content_set, (EPRP_GETTER *)ecn_data_prop_get });

/* メーカーコード */
ECN_DEF_EPRP (LOCAL_NODE_EOBJ, { 0x8A, EPC_RULE_GET, 3, (intptr_t)&local_node_data.manufacturer_code,
(EPRP_SETTER *)ecn_data_prop_set, (EPRP_GETTER *)ecn_data_prop_get });
```

第一引数は、自ノードのプロパティを定義していることを示すため、自ノードの識別子「LOCAL_NODE_EOBJ」を入れます。第二引数には EPC の値、第三引数にはアクセスルールと状態変化時アナウンスを設定します。アクセスルールの Set は「EPC_RULE_SET」、Get は「EPC_RULE_GET」、状態変化時アナウンスは「EPC_ANNOUNCE」の定義を使用し、複数ある場合は OR 演算子「|」でつなげます。第四引数はプロパティのサイズをバイト単位で指定します。

第五引数以降の設定については後述します。

【5】　自ノードのプロパティ保存領域の構造体定義

（ア）　プロパティの中にある構造体の定義

プロパティ名称	EPC	プロパティ内容 値域	データ型	データサイズ(Byte)	アクセスルール	必須	状態変化時アナウンス	
動作状態	0x80	ノードの動作状態を示す。 0x30=起動中,0x31=未起動中	unsigned char	1	Set Get	 ○	○	
Version情報	0x82	通信ミドルウェアが適用しているECHONET LiteのVersion、および通信ミドルウェアがサポートする電文タイプを示す。 1バイト目：メジャーバージョン（小数点以上）をBinaryで示す。 2バイト目：マイナーバージョン（小数点以上）をBinaryで示す。 3、4バイト目：電文タイプをビットマップで示す。	unsigned char× 4	4	Get	○		
識別番号	0x83	オブジェクトを、ドメイン内で一意に識別するための番号。 1バイト目：下位通信層IDフィールド 0x01～0xFD：下位通信層で使用される通信プロトコルで固有の番号が振られている場合、プロトコル種別に応じて、 任意に設定（ECHONET Liteでは使用しない） 0xFE:2～17バイトをメーカ規定形式により設定 0xFF:2～9バイトを乱数により生成するプロトコルを下位通信層で使用する場合に設定 0x00:識別番号未設定 2バイト目以降：固有番号フィールド	unsigned char×9 or unsigned char×17	9 or 17	Get	○		
異常内容	0x89	異常内容 0x0000～0x03E8(0～1000)	unsigned short	2	Get			
メーカーコード	0x8A	3バイトで指定 (ECHONETコンソーシアムで規定。)	unsigned char× 3	3	Get	○		

ノードプロファイルクラスの中には、「Version 情報」と「識別番号」というプロパティがありますが、このプロパティは単純な数値ではなく、数値を組み合わせた C 言語でいうところの構造体のようになっていますので、保存領域も構造体とした方が、仕様とコードの差異が少なくなるので、構造体として定義します。

（イ）　Version 情報の構造体

```
/*
 * Ｖｅｒｓｉｏｎ情報の型
 */
struct version_information_t {
        /* メジャーバージョン(小数点以上) */
        uint8_t major_version_number;
        /* マイナーバージョン(小数点以下) */
        uint8_t minor_version_number;
        /* 電文タイプ */
        uint8_t message_type[2];
};
```

規格書のプロパティ内容／値域の項目に、「１バイト目：…」とうい記述どおりに構造体を作成します。

（ウ）　識別番号の構造体

```
/*
 * メーカーコードの型
 */
struct manufacturer_code_t {
        /* メーカーコード */
        uint8_t manufacturer_code[3];
};

/*
 * 識別番号の型
 */
struct node_identification_number_t {
        /* 下位通信層IDフィールド */
        uint8_t lower_communication_id_field;
        /* メーカーコード */
        struct manufacturer_code_t manufacturer_code;
        /* ユニークID部(メーカー独自) */
        uint8_t unique_id_section[13];
};
```

識別番号の領域サイズは２種類ありますが、今回は１７バイトの方で定義しました。

（エ）　自ノードの構造体定義

プロパティ名称	EPC	プロパティ内容 値域	データ型	データサイズ(Byte)	アクセスルール	必須	状態変化時アナウンス	
動作状態	0x80	ノードの動作状態を示す。 0x30=起動中,0x31=未起動中	unsigned char	1	Set Get	 ○	○	
Version情報	0x82	通信ミドルウェアが適用しているECHONET LiteのVersion、および通信ミドルウェアがサポートする電文タイプを示す。 1バイト目：メジャーバージョン（小数点以上）をBinaryで示す。 2バイト目：マイナーバージョン（小数点以上）をBinaryで示す。 3、4バイト目：電文タイプをビットマップで示す。	unsigned char× 4	4	Get	○		
識別番号	0x83	オブジェクトを、ドメイン内で一意に識別するための番号。 1バイト目：下位通信層IDフィールド 0x01～0xFD：下位通信層で使用される通信プロトコルで固有の番号が振られている場合、プロトコル種別に応じて、 任意に設定（ECHONET Liteでは使用しない） 0xFE:2～17バイトをメーカ規定形式により設定 0xFF:2～9バイトを乱数により生成するプロトコルを下位通信層で使用する場合に設定 0x00:識別番号未設定 2バイト目以降：固有番号フィールド	unsigned char×9 or unsigned char×17	9 or 17	Get	○		
異常内容	0x89	異常内容 0x0000～0x03E8(0～1000)	unsigned short	2	Get			
メーカーコード	0x8A	3バイトで指定 (ECHONETコンソーシアムで規定。)	unsigned char× 3	3	Get	○		

プロパティのデータ型とデータサイズを元に、自ノードのプロパティ保存領域用の構造体を定義します。

```
/*
 *  ノードプロファイルオブジェクト
 */
struct node_profile_object_t {
        /* 動作状態 */
        uint8_t operation_status;
        /* Ｖｅｒｓｉｏｎ情報 */
        struct version_information_t version_information;
        /* 識別番号 */
        struct node_identification_number_t identification_number;
        /* 異常内容 */
        uint16_t fault_content;
        /* メーカーコード */
        struct manufacturer_code_t manufacturer_code;
};
```

Version 情報と識別番号は、先ほど作成した構造体を使用します。

【６】　一般照明の cfg ファイル定義

今回選択した一般照明のプロパティを下の表に規格書から転記しました。

プロパティ名称	EPC	プロパティ内容 値域	データ型	データサイズ(Byte)	アクセスルール	必須	状態変化時アナウンス
動作状態	0x80	ON／OFFの状態を示す。 0x30=起動中,0x31=未起動中	unsigned char	1	Set Get	○ ○	○
点灯モード設定	0xB6	自動／通常灯／常夜灯／カラー灯 0x41=自動,0x42=通常灯,0x43=常夜灯,0x45=カラー灯	unsigned char	1	Set/Get		
設置場所	0x81	設置場所を示す。 0x41=異常発生有,0x42=異常発生無	unsigned char	1or17	Set/Get	○	
規格Version情報	0x82	対応するAPPENDIXのリリース番号を示す。 1バイト目：0x00固定(for future reserved) 2バイト目：0x00固定(for future reserved) 3バイト目：リリース順をASCIIで示す。 4バイト目：0x00固定(for future reserved)	unsigned char×4	4	Get	○	
異常発生状態	0x88	何らかの異常の発生状況を示す。 0x41=異常発生有,0x42=異常発生無	unsigned char	1	Get	○	
メーカーコード	0x8A	3バイトで指定 (ECHONETコンソーシアムで規定。)	unsigned char× 3	3	Get	○	

一般照明の ID の識別子を「GENERAL_LIGHTING_CLASS_EOBJ」とし、静的 API「ECN_CRE_EOBJ」の第一引数に設定します。

自ノードのIDの識別子：LOCAL_NODE_EOBJ
一般照明のIDの識別子：GENERAL_LIGHTING_CLASS_EOBJ

```
/*
 *  一般照明クラス
 */
ECN_CRE_EOBJ (GENERAL_LIGHTING_CLASS_EOBJ, { EOBJ_DEVICE, LOCAL_NODE_EOBJ, 0, EOJ_X1_AMENITY,
EOJ_X2_GENERAL_LIGHTING_CLASS, EOJ_X3_GENERAL_LIGHTING_CLASS });

/* 動作状態 */
ECN_DEF_EPRP (GENERAL_LIGHTING_CLASS_EOBJ, { 0x80, EPC_ANNOUNCE | EPC_RULE_SET | EPC_RULE_GET, 1,
(intptr_t)&general_lighting_class_data.property80, (EPRP_SETTER *)onoff_prop_set, (EPRP_GETTER *)ecn_data_prop_get
});

/* 点灯モード設定 */
ECN_DEF_EPRP (GENERAL_LIGHTING_CLASS_EOBJ, { 0xB6, EPC_RULE_SET | EPC_RULE_GET, 1,
(intptr_t)&general_lighting_class_data.propertyB6, (EPRP_SETTER *)lighting_mode_prop_set, (EPRP_GETTER
*)ecn_data_prop_get });

/* 設置場所 */
ECN_DEF_EPRP (GENERAL_LIGHTING_CLASS_EOBJ, { 0x81, EPC_ANNOUNCE | EPC_RULE_SET | EPC_RULE_GET, 1,
(intptr_t)&general_lighting_class_data.property81, (EPRP_SETTER *)ecn_data_prop_set, (EPRP_GETTER
*)ecn_data_prop_get });

/* 規格Ｖｅｒｓｉｏｎ情報 */
ECN_DEF_EPRP (GENERAL_LIGHTING_CLASS_EOBJ, { 0x82, EPC_RULE_GET, 4,
(intptr_t)&general_lighting_class_data.property82, (EPRP_SETTER *)NULL, (EPRP_GETTER *)ecn_data_prop_get });

/* 異常発生状態 */
ECN_DEF_EPRP (GENERAL_LIGHTING_CLASS_EOBJ, { 0x88, EPC_ANNOUNCE | EPC_RULE_GET, 1,
(intptr_t)&general_lighting_class_data.property88, (EPRP_SETTER *)alarm_prop_set, (EPRP_GETTER *)ecn_data_prop_get
});

/* メーカーコード */
ECN_DEF_EPRP (GENERAL_LIGHTING_CLASS_EOBJ, { 0x8A, EPC_RULE_GET, 3,
(intptr_t)&general_lighting_class_data.property8A, (EPRP_SETTER *)ecn_data_prop_set, (EPRP_GETTER
*)ecn_data_prop_get });
```

第二引数には機器オブジェクトを示す「EOBJ_DEVICE」を設定し、第三引数には自ノードの配下であることを示すため「LOCAL_NODE_EOBJ」を設定します。第四引数には「住宅・設備関連機器」クラスグループコードを示す「EOJ_X1_AMENITY」、第五引数には「一般照明」クラスコードを示す「EOJ_X2_GENERAL_LIGHTING_CLASS」、第六引数はユーザーが定義するインスタンスコードを示す「EOJ_X3_GENERAL_LIGHTING_CLASS」を設定します。

【７】　一般照明のプロパティ保存領域

プロパティ名称	EPC	プロパティ内容 値域	データ型	データサイズ(Byte)	アクセスルール	必須	状態変化時アナウンス	
動作状態	0x80	ON／OFFの状態を示す。 0x30=起動中,0x31=未起動中	unsigned char	1	Set Get	○ ○	○	
点灯モード設定	0xB6	自動／通常灯／常夜灯／カラー灯 0x41=自動,0x42=通常灯,0x43=常夜灯,0x45=カラー灯	unsigned char	1	Set/Get			
設置場所	0x81	設置場所を示す。 0x41=異常発生有,0x42=異常発生無	unsigned char	1or17	Set/Get	○		
規格Version情報	0x82	対応するAPPENDIXのリリース番号を示す。 1バイト目：0x00固定(for future reserved) 2バイト目：0x00固定(for future reserved) 3バイト目：リリース順をASCIIで示す。 4バイト目：0x00固定(for future reserved)	unsigned char×4	4	Get	○		
異常発生状態	0x88	何らかの異常の発生状況を示す。 0x41=異常発生有,0x42=異常発生無	unsigned char	1	Get	○		
メーカーコード	0x8A	3バイトで指定 (ECHONETコンソーシアムで規定。)	unsigned char× 3	3	Get	○		

機器オブジェクトスーパークラスの中には「規格 Version 情報」というプロパティがありますが、ノードプロファイルクラスの「Version 情報」と同様に構造体として定義します。

```
/*
 * 規格Ｖｅｒｓｉｏｎ情報の型
 */
struct standard_version_information_t {
        /* 固定1(for future reserved) */
        uint8_t reserved1;
        /* 固定2(for future reserved) */
        uint8_t reserved2;
        /* リリース順をASCIIで示す */
        uint8_t order_of_release;
        /* 固定4(for future reserved) */
        uint8_t reserved4;
};
```

プロパティのデータ型とデータサイズを元に、一般照明のプロパティ保存領域用構造体を定義します。

```
/*
 * 一般照明クラス
 */
struct general_lighting_t {
        /* 動作状態 */
        uint8_t operation_status;
        /* 点灯モード設定 */
        uint8_t lighting_mode_setting;
        /* 設置場所 */
        uint8_t installation_location;
        /* 規格Ｖｅｒｓｉｏｎ情報 */
        struct standard_version_information_t standard_version_information;
        /* 異常発生状態 */
        uint8_t fault_status;
        /* メーカーコード */
        struct manufacturer_code_t manufacturer_code;
};
```

1.3. アプリケーション スケルトン生成ツール

ここまでに説明した機器オブジェクトの定義を GUI の操作で作成するツール、「TOPPERS/ECNL アプリケーション スケルトン生成ツール」の使い方を説明します。

このツールを使用すると、機器オブジェクトの定義名の入力、プロパティの選択をすることで、main.c、main.cfg、main.h のスケルトンコードを生成することができます。

2015 年 2 月 23 日現在、プロパティの一部が未対応のβ版ですが、多くの機器オブジェクトのプロパティ定義の手間を簡略化することができると思います。

ツールを使用しないで機器オブジェクトの定義を行う場合は、この章は読む必要はありません。

【1】 動作環境

OS は、Windows 7 と Windows 8.1 で動作することを確認しています。アプリケーション内に Web ブラウザ埋め込んでいますので、Internet Explorer 11 が必要です。また、ライブラリに.NET Framework 4.5 を使用しています。

OS	Windows 7/8.1
依存ソフトウェア	Internet Explorer 11
	.NET Framework 4.5

【2】 インストール

以下のサイトから実行ファイルの ecnl_sklgen-1.B.0.zip をダウンロードします。

https://www.toppers.jp/ecnl-download.html

圧縮ファイルを適当なフォルダに展開すると、以下のファイルが展開されます。

```
EcnlSklGen.exe
EcnlSklGen.exe.config
EcnlSklGenRes.dll
```

EcnlSklGen.exe が実行ファイル、EcnlSklGen.exe.config が初期設定ファイル、EcnlSklGenRes.dll が実行ファイルに必要な Web コンテンツの入ったリソース DLL になります。

【3】 アンインストール

展開したファイルを削除してください。レジストリやテンポラリフォルダなどは使用していません。

【４】　起動

EcnlSklGen.exe をエクスプローラからダブルクリックして起動します。
起動直後の画面は以下のようになります。

上から順に各部の説明をします。タイトルの下のインスタンス部分には、定義中のノードと機器オブジェクトがリスト表示されます。「スケルトン生成」ボタンは、機器オブジェクトの定義が終わり、スケルトンコードを生成したいときに使用します。

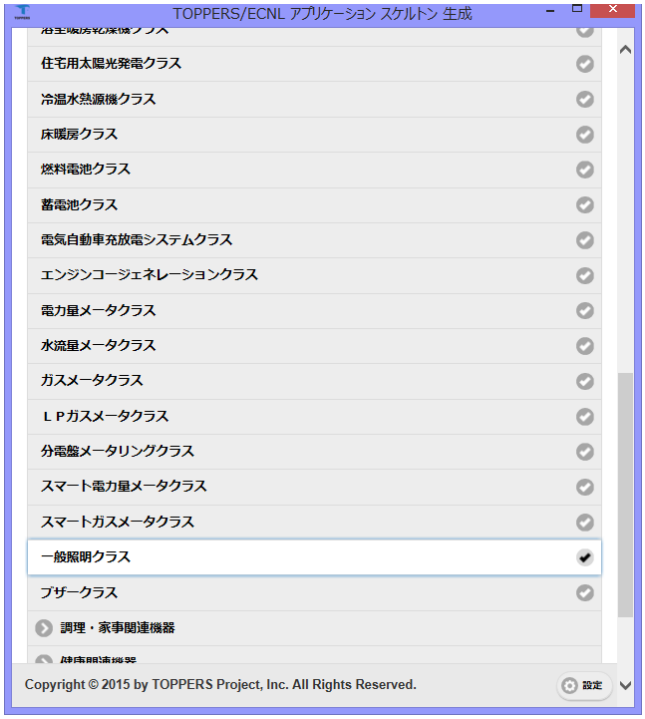
ノード・オブジェクト・プロパティが並んでいるタブで、選択によってタブ以下に表示されている部分が変わります。起動直後は、オブジェクトが選択されています。

自ノードとしてあらかじめ「LOCAL_NODE_EOBJ」が定義されていますので、一般照明の機器オブジェクトを１つ定義します。

【5】　機器オブジェクトのクラス選択

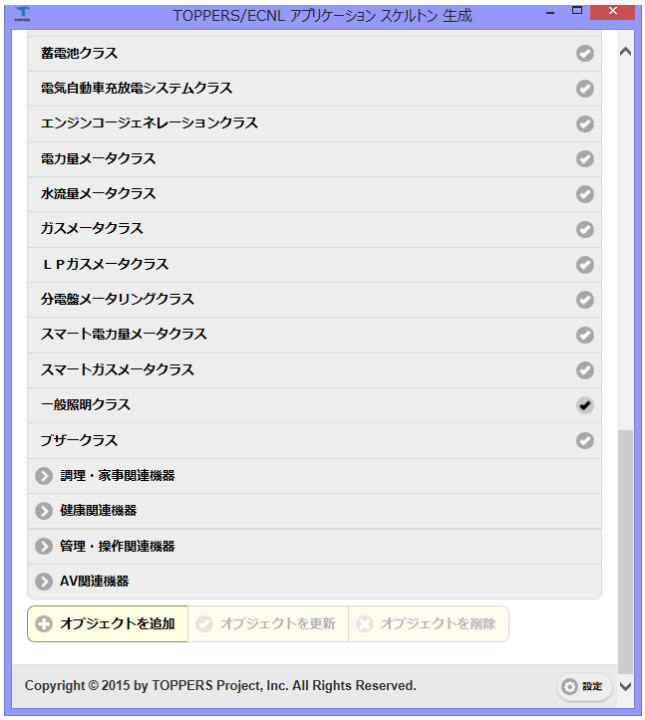
一般照明クラスを下のクラスグループごとに折りたたまれたリストから、選択します。

一般照明クラスは、住宅・設備関連機器クラスグループの中にありますので、クリックして展開します。



展開して一般照明にチェックを入れます。

一般照明の機器オブジェクトの定義名を入力します。クラスを選択したときに、クラス名から機器オブジェクトの定義名を作成したものが入っているので、インスタンスが複数あるなどの場合は変更します。



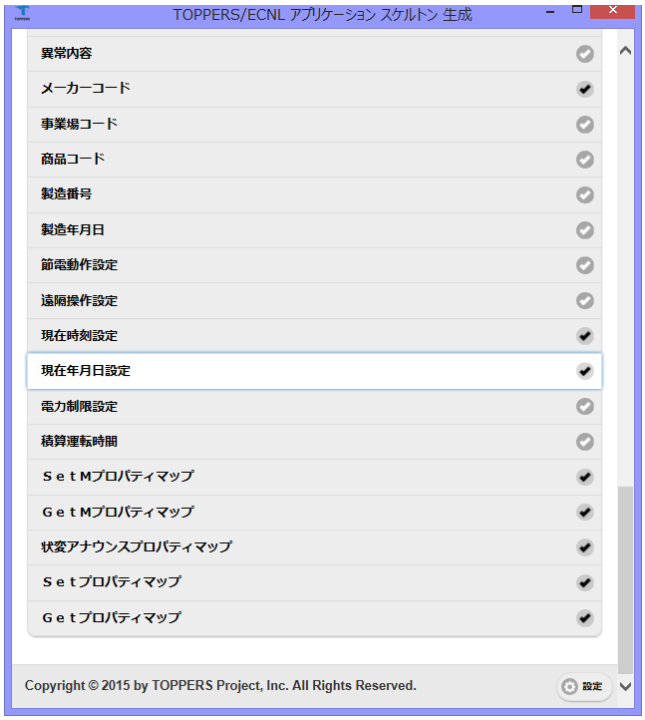
ページの最後に、「オブジェクトを追加」を押して、一般照明を追加します。

インスタンスのリスト部分の自ノード「LOCAL_NODE_EOBJ」の中に、一般照明「GENERAL_LIGHTING_CLASS_EOBJ」が表示されていれば、機器オブジェクトの作成は完了です。

【6】　プロパティの選択

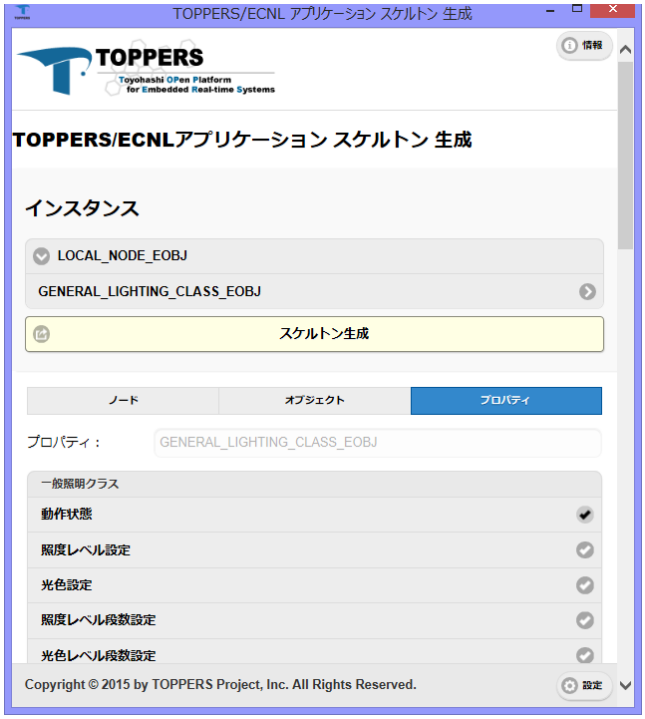
インスタンスのリスト部分の一般照明「GENERAL_LIGHTING_CLASS_EOBJ」をクリックし、タブの「プロパティ」をクリックします。

この画面で、一般照明のプロパティを選択することができます。

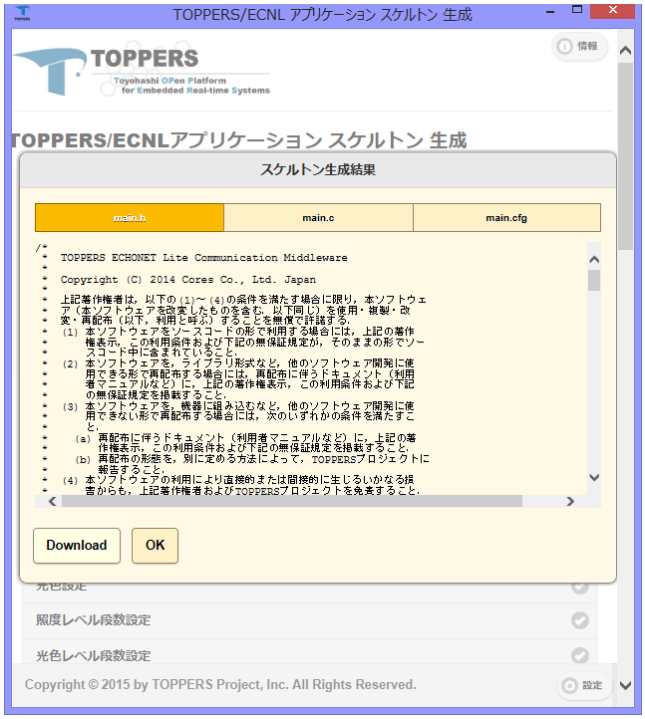


必須項目はすでに選択されているので、追加で「現在時刻設定」と「現在年月日設定」をクリックし、チェックを入れます。

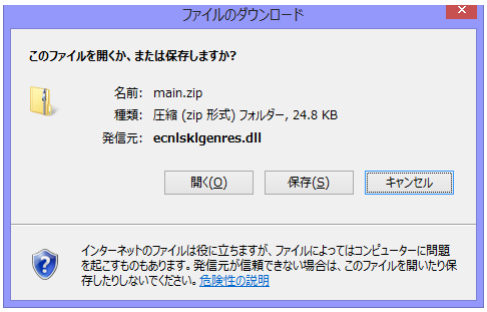
【7】　スケルトンの生成



機器オブジェクトの定義とプロパティの選択が出来たら、ページ上方の「スケルトン生成」ボタンをクリックします。



ダイアログに main.c、main.cfg、main.h のスケルトンコードが表示されます。コードが問題なければ「Download」ボタンでダウンロードします。



ダウンロードするファイルは、3 つのスケルトンコードをまとめた main.zip でダウンロードできます。

プロパティのコールバック関数などの実装を行い、スケルトンコードを完成させます。

1.4. プロパティのコールバック関数の定義

プロパティの設定または取得時にミドルウェアから呼び出される、コールバック関数の実装方法について説明します。

【1】　動作状態プロパティ

cfg ファイルの「ECN_DEF_EPRP」の動作状態の記述で、第五引数以降を見ていきます。

```
/* 動作状態 */
ECN_DEF_EPRP (GENERAL_LIGHTING_CLASS_EOBJ, { 0x80, EPC_RULE_SET | EPC_RULE_GET | EPC_ANNOUNCE, 1,
(intptr_t)&general_lighting_class_data.operation_status, (EPRP_SETTER *)onoff_prop_set, (EPRP_GETTER
*)ecn_data_prop_get });
```

第五引数は拡張情報ですが、これを使って設定・取得コールバック関数の処理でプロパティの保存領域を参照するため、拡張情報に保存領域の先頭アドレスを設定しています。第六引数に設定コールバック関数、第七引数に取得コールバック関数を設定します。

```
/*
 * 一般照明クラス
 */
struct general_lighting_t {
        /* 動作状態 */
        uint8_t operation_status;
        /* 点灯モード設定 */
        uint8_t lighting_mode_setting;
        /* 設置場所 */
        uint8_t installation_location;
        /* 規格Ｖｅｒｓｉｏｎ情報 */
        struct standard_version_information_t standard_version_information;
        /* 異常発生状態 */
        uint8_t fault_status;
        /* メーカーコード */
        struct manufacturer_code_t manufacturer_code;
};
```

保存領域の実体は c ファイルに定義します。電源起動時または、リセット時に値が初期化されるよう設定しておきます。

```
extern struct general_lighting_t general_lighting_class_data;      /* 一般照明クラスのデータ */

/* 一般照明クラス */
struct general_lighting_t general_lighting_class_data = {
        0x30,       /* 動作状態 */
        0x41,       /* 点灯モード設定 */
        0x00,       /* 設置場所 */
        { 0x00, 0x00, 'C', 0x00 },        /* 規格Ｖｅｒｓｉｏｎ情報 */
        0x41,       /* 異常発生状態 */
        { MAKER_CODE },    /* メーカーコード */
};
```

（ア）　取得コールバック関数

今回定義している動作状態プロパティでは、プロパティ保存領域を作成したので、取得コールバックについてはミドルウェアから用意される「ecn_data_prop_get」を使用して、プロパティサイズを元にプロパティ保存領域から値を返すようにしています。この関数では、拡張情報に取得するプロパティの値の先頭アドレスが設定されている必要がありますので、cfg ファイルの定義もそのように記述しています。

（イ）　設定コールバック関数

設定コールバックでは、7 セグ LED の表示を変更するためコールバック関数を定義

します。

プロパティ名称	EPC	プロパティ内容 値域	データ型	データサイズ(Byte)	アクセスルール	必須	状態変化時アナウンス	
動作状態	0x80	ON／OFFの状態を示す。 0x30=起動中,0x31=未起動中	unsigned char	1	Set Get	○ ○	○	
点灯モード設定	0xB6	自動／通常灯／常夜灯／カラー灯 0x41=自動,0x42=通常灯,0x43=常夜灯,0x45=カラー灯	unsigned char	1	Set/Get			
設置場所	0x81	設置場所を示す。 0x41=異常発生有,0x42=異常発生無	unsigned char	1or17	Set/Get	○		
規格Version情報	0x82	対応するAPPENDIXのリリース番号を示す。 1バイト目：0x00固定(for future reserved) 2バイト目：0x00固定(for future reserved) 3バイト目：リリース順をASCIIで示す。 4バイト目：0x00固定(for future reserved)	unsigned char×4	4	Get	○		
異常発生状態	0x88	何らかの異常の発生状況を示す。 0x41=異常発生有,0x42=異常発生無	unsigned char	1	Get	○		
メーカーコード	0x8A	3バイトで指定 (ECHONETコンソーシアムで規定。)	unsigned char× 3	3	Get	○		

規格書にある動作状態のプロパティ内容／値域の記述を見ると、0x30 が照明の ON で、0x31 が照明の OFF と書かれていますので、コールバック関数ではこの値の時だけ処理を行うように記述します。

ミドルウェアが受け取った電文を処理する中で、静的 API「ECN_DEF_EPRP」で定義したプロパティを変更することになったら、設定コールバック関数を呼び出します。第一引数には、静的 API「ECN_DEF_EPRP」で定義した情報が入っている「EPRPINIB」構造体の先頭アドレスが入っていますので、設定した拡張情報を取得したい時などに使用します。第二引数には、受け取ったプロパティ値の先頭アドレス、第三引数には受け取ったプロパティのサイズ、第四引数には通知を行うかどうかを設定するためのフラグの先頭番地が入ります。

```
/*
 * 動作状態ON/OFF設定関数（0x30, 0x31のみ受け付け）
 */
int onoff_prop_set(const EPRPINIB *item, const void *src, int size, bool_t *anno)
{
	/* サイズが 1 以外は受け付けない */
	if(size != 1)
		return 0;

	*anno = *((uint8_t*)item->exinf) != *((uint8_t*)src);

	switch(*(uint8_t *)src){
	/* ONの場合 */
	case 0x30:
		*((uint8_t *)item->exinf) = *((uint8_t *)src);
		/* 7segの"."をON */
		sil_wrb_mem((uint8_t *)0xFFFFF412, sil_reb_mem((uint8_t *)0xFFFFF412) & ~0x80);
		break;
	/* OFFの場合 */
	case 0x31:
		*((uint8_t *)item->exinf) = *((uint8_t *)src);
		/* 7segの"."をOFF */
		sil_wrb_mem((uint8_t *)0xFFFFF412, sil_reb_mem((uint8_t *)0xFFFFF412) | 0x80);
		break;
	/* 0x30か0x31以外は受け付けない */
	default:
		return 0;
	}

	return 1;
}
```

まず、引数で渡されたサイズが正しいか確認します。サイズが正しくなかった場合は、戻り値に 0 を設定することで、ミドルウェア側でプロパティ設定失敗と処理します。

次に、引数で渡されたプロパティ値の設定処理を行います。0x30 か 0x31 以外の場合は、戻り値を 0 に設定しプロパティ設定失敗とします。正しい値の場合は、プロパティ保存領域に値を保存して、 7 セグ LED のドットの点灯／消灯を行います。戻り値は、プロパティのサイズ 1 を返すことで、ミドルウェア側でプロパティ設定成功として処理します。

第四引数の説明を飛ばしましたが、状態変化時アナウンス属性を持つプロパティは、その名の通り、状態が変化した時に「プロパティ通知」電文を送信しなければなりません。第四引数は状態変化が起きたかどうかを設定する引数となっています。プロパティの保存値（現在値）と設定値が違っている場合は true に設定して、状態変化があったことをミドルウェアに知らせます。

ちなみに、この引数を true としても、プロパティ設定が成功しなければ「プロパティ通知」電文を送信しませんし、状態変化時アナウンス属性を持っていない場合も送信しません。

【２】　点灯モード設定プロパティ

cfg ファイルの点灯モードの定義を見ます。

```
/* 点灯モード設定 */
ECN_DEF_EPRP (GENERAL_LIGHTING_CLASS_EOBJ, { 0xB6, EPC_RULE_SET | EPC_RULE_GET, 1,
(intptr_t)&general_lighting_class_data.lighting_mode_setting, (EPRP_SETTER
*)general_lighting_lighting_mode_setting_set, (EPRP_GETTER *)ecn_data_prop_get });
```

動作状態と同様に、拡張情報にプロパティの保存領域の先頭アドレスを設定します。

```
/*
 *  一般照明クラス
 */
struct general_lighting_t {
        /* 動作状態 */
        uint8_t operation_status;
        /* 点灯モード設定 */
        uint8_t lighting_mode_setting;
        /* 設置場所 */
        uint8_t installation_location;
        /* 規格Ｖｅｒｓｉｏｎ情報 */
        struct standard_version_information_t standard_version_information;
        /* 異常発生状態 */
        uint8_t fault_status;
        /* メーカーコード */
        struct manufacturer_code_t manufacturer_code;
};
```

プロパティ保存領域の実体には、初期値を設定しておきます。

```
extern struct general_lighting_t general_lighting_class_data;      /* 一般照明クラスのデータ */

/* 一般照明クラス */
struct general_lighting_t general_lighting_class_data = {
        0x30,       /* 動作状態 */
        0x41,       /* 点灯モード設定 */
        0x00,       /* 設置場所 */
        { 0x00, 0x00, 'C', 0x00 },        /* 規格Ｖｅｒｓｉｏｎ情報 */
        0x41,       /* 異常発生状態 */
        { MAKER_CODE },    /* メーカーコード */
};
```

（ア）　取得コールバック関数

プロパティ取得時には特に処理はないので、値を返す処理を行う「ecn_data_prop_get」を設定します。

（イ）　設定コールバック関数

点灯モード設定プロパティの場合も、動作モードプロパティと同様に、設定コールバック関数を定義します。

プロパティ名称	EPC	プロパティ内容 値域	データ型	データサイズ(Byte)	アクセスルール	必須	状態変化時アナウンス	
動作状態	0x80	ON／OFFの状態を示す。 0x30=起動中,0x31=未起動中	unsigned char	1	Set Get	○ ○	○	
点灯モード設定	0xB6	自動／通常灯／常夜灯／カラー灯 0x41=自動,0x42=通常灯 0x43=常夜灯,0x45=カラー灯	unsigned char	1	Set/Get			
設置場所	0x81	設置場所を示す。 0x41=異常発生有,0x42=異常発生無	unsigned char	1or17	Set/Get	○		
規格Version情報	0x82	対応するAPPENDIXのリリース番号を示す。 1バイト目：0x00固定(for future reserved) 2バイト目：0x00固定(for future reserved) 3バイト目：リリース順をASCIIで示す。 4バイト目：0x00固定(for future reserved)	unsigned char×4	4	Get	○		
異常発生状態	0x88	何らかの異常の発生状況を示す。 0x41=異常発生有,0x42=異常発生無	unsigned char	1	Get	○		
メーカーコード	0x8A	3バイトで指定 (ECHONETコンソーシアムで規定。)	unsigned char× 3	3	Get	○		

点灯モード設定プロパティは、値域して自動、通常灯、常夜灯、カラー灯のそれぞれ 0x41、0x42、0x43、0x45 の値のみをとるプロパティなので、コールバック関数の定義でも、この値の時のみ処理するように実装します。動作状態プロパティの switch 文の case を 4 つに増やしたようになります。

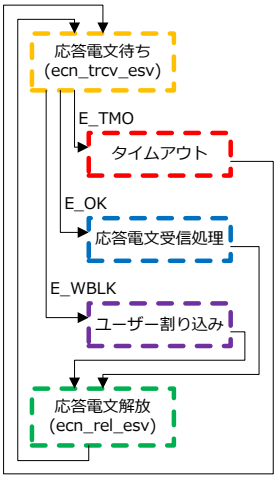
```
/*
 * 点灯モード設定設定関数
 */
int general_lighting_lighting_mode_setting_set(const EPRPINIB *item, const void *src, int size, bool_t *anno)
{
	uint8_t sseg = sil_reb_mem((uint8_t *)0xFFFFF412) & 0x80;

	/* サイズが1以外は受け付けない */
	if(size != 1)
		return 0;

	switch(*(uint8_t *)src){
	/* 自動の場合 */
	case 0x41:
		*((uint8_t *)item->exinf) = *((uint8_t *)src);
		/* 7segを"A"と表示 */
		sil_wrb_mem((uint8_t *)0xFFFFF412, sseg | 0x08);
		break;
	/* 通常灯の場合 */
	case 0x42:
		*((uint8_t *)item->exinf) = *((uint8_t *)src);
		/* 7segを"b"と表示 */
		sil_wrb_mem((uint8_t *)0xFFFFF412, sseg | 0x03);
		break;
	/* 常夜灯の場合 */
	case 0x43:
		*((uint8_t *)item->exinf) = *((uint8_t *)src);
		/* 7segを"C"と表示 */
		sil_wrb_mem((uint8_t *)0xFFFFF412, sseg | 0x46);
		break;
	/* カラー灯の場合 */
	case 0x45:
		*((uint8_t *)item->exinf) = *((uint8_t *)src);
		/* 7segを"E"と表示 */
		sil_wrb_mem((uint8_t *)0xFFFFF412, sseg | 0x06);
		break;
	default:
		/* 上記以外は受け付けない */
		return 0;
	}

	return 1;
}
```

1.5. メインタスクの定義

アプリケーションの動作を行うメインタスクの定義について説明します。

【1】　メインタスクのメインルーチン

```
/*
 * メインタスク
 */
void main_task(intptr_t exinf)
{
	ER ret, ret2;
	SYSTIM prev, now;
	TMO timer;
	T_EDATA *esv;
	uint8_t brkdat[64];
	int32_t len;

	/* TINETが起動するまで待つ */
	ret = tslp_tsk(1000);
	if(ret != E_TMOUT)
		return;

	/* ECHONETミドルウェアを起動 */
	ret = act_tsk(ECN_UDP_TASK);
	if(ret != E_OK)
		return;

	/* 初期化 */
	main_initialize();

	ret2 = get_tim(&now);
	if (ret2 != E_OK){
		syslog(LOG_ERROR, "get_tim");
		return;
	}

	for(;;){
		prev = now;

		/* タイマー取得 */
		timer = main_get_timer();

		/* 応答電文待ち */
		ret = ecn_trcv_esv(&esv, timer);
		if ((ret != E_OK) && (ret != E_WBLK) && (ret != E_TMOUT)){
			syslog(LOG_ERROR, "ecn_trcv_esv");
			break;
		}

		ret2 = get_tim(&now);
		if (ret2 != E_OK){
			syslog(LOG_ERROR, "get_tim");
			break;
		}

		/* 時間経過 */
		main_progress(now - prev);

		/* Echonet電文受信の場合 */
		if (ret == E_OK) {
			/* Echonet電文受信処理 */
			main_recv_esv(esv);

			/* 領域解放 */
			ecn_rel_esv(esv);
		}
		/* 応答電文待ちの割り込みの場合 */
		else if (ret == E_WBLK) {
			/* 応答電文待ちの割り込みデータ取得 */
			ecn_get_brk_dat(esv, brkdat, sizeof(brkdat), &len);

			/* 応答電文待ちの割り込み処理 */
			main_break_wait(brkdat, len);

			/* 領域解放 */
			ecn_rel_esv(esv);
		}
		/* タイムアウト処理 */
		main_timeout();
	}
}
```



処理のブロック図とソースコードの対応を色分けしました。

応答電文待ちは、API「ecn_trcv_esv」でタイムアウト付きで待ちます。呼のメインルーチンでは、「main_get_timer」関数でタイムアウト時間を取得しています。また、「main_progress」関数で時刻の経過を計算し、次のタイムアウト時間の計算に使用しています。タイムアウト処理は「main_timeout」関数で行っています。この処理については後述します。

ECHONET 電文を受信した場合は、API「ecn_trcv_esv」の戻り値が E_OK となりますので、「main_recv_esv()」関数で処理します。今回のアプリケーションでは処理はありません。

ユーザーが応答電文待ちに割り込んで待ちを解除した場合、API「ecn_trcv_esv」の戻り値が E_WBLK となりますので、「main_break_wait()」関数で処理します。今回のアプリケーションでは処理はありません。

ECHONET 電文を受信した場合と、ユーザーが応答電文待ちに割り込んで待ちを解除した場合は、受信した電文の領域解放を行う必要があります。

【2】　タイムアウト処理

```
/*
 * タイムアウト処理
 */
static void main_timeout()
{
	uint8_t btn;

	if(main_timer != 0)
		return;

	switch(main_state){
	case main_state_idle:
		/* 10ms後にボタン状態を確認 */
		main_timer = 10;

		/* ボタン状態読み込み */
		btn = sil_reb_mem((uint8_t *)0xFFFFF413);

		/* ボタン１の処理 */
		if(((btn & 0x01) == 0) && !main_btn1_state){
			main_btn1_count++;
			if(main_btn1_count > 10){
				main_btn1_count = 0;
				main_btn1_state = true;

				main_btn1_change(true);
			}
		}
		else if(((btn & 0x01) != 0) && main_btn1_state){
			main_btn1_count++;
			if(main_btn1_count > 10){
				main_btn1_count = 0;
				main_btn1_state = false;

				main_btn1_change(false);
			}
		}
		else{
			main_btn1_count = 0;
		}

		/* ボタン２の処理 */
		if(((btn & 0x10) == 0) && !main_btn2_state){
			main_btn2_count++;
			if(main_btn2_count > 10){
				main_btn2_count = 0;
				main_btn2_state = true;

				main_btn2_change(true);
			}
		}
		else if(((btn & 0x10) != 0) && main_btn2_state){
			main_btn2_count++;
			if(main_btn2_count > 10){
				main_btn2_count = 0;
				main_btn2_state = false;

				main_btn2_change(false);
			}
		}
		else{
			main_btn2_count = 0;
		}
		break;
	}
}
```


タイムアウト処理では、「main_progress」関数で計算された、「main_timer」変数が 0 の時に処理を行います。この変数にはタイマー値を[ms]単位で設定することで、タイムアウト処理を行うことが出来るように設計しています。

「main_state」は、今回のアプリケーションでは、「main_state_idle」以外の値は取りません。

このタイムアウト処理で、10ms ごとにボタン 1 とボタン 2 の押下状態を確認します。チャタリング防止のため 10ms×10 回連続して状態が変わった時に、ボタンの状態を変更するように処理しています。

ボタンの状態は「main_btn1_state」変数と「main_btn2_state」変数とに保持し

ています。ポートから読み出した値が保持している値と変化した時、カウンターを加算し 10 回になった時にカウンターをクリアし、保持している状態を変化させ、各サブルーチンを呼びます。

ポートから読み出した値と、保持した状態が同じときはカウンターをクリアしてやり直します。

状態変化時に呼び出す関数は、それぞれ「main_btn1_change」と「main_btn2_change」で、引数に ON に変換したか OFF に変化したかの引数を持ちます。

【３】　ボタン１状態変化処理

```
bool_t main_on = false;

/*
 * ボタン１状態変化処理
 */
static void main_btn1_change(bool_t push)
{
	ER ret;
	T_EDATA *esv;
	uint8_t p_edt[1];

	/* 押されて戻った時に処理する */
	if(push)
		return;

	/* ON/OFF状態の切り替え */
	main_on = !main_on;
	p_edt[0] = main_on ? 0x30 : 0x31;

	/* プロパティ設定電文作成 */
	ret = ecn_esv_setc(&esv, GENERAL_LIGHTING_CLASS_EOBJ, 0x80, 1, p_edt);
	if(ret != E_OK){
		syslog(LOG_ERROR, "ecn_esv_setc");
		return;
	}

	/* 電文送信 */
	ecn_snd_esv(esv);
}
```

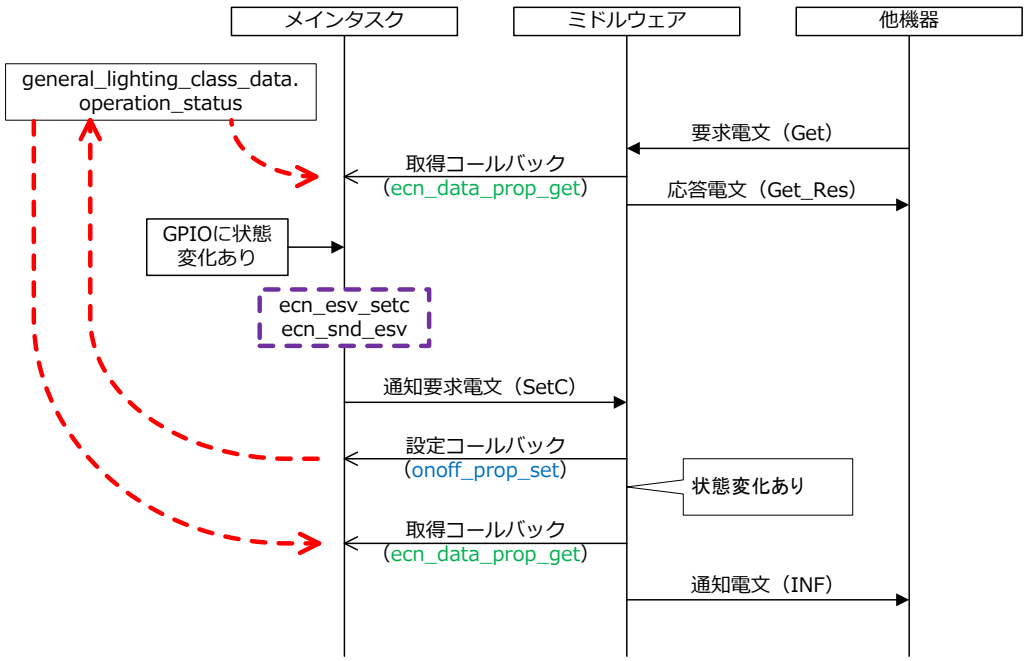
ボタン１の状態変化時は、動作状態プロパティの ON/OFF を切り替える処理を行います。

設定コールバック関数を実装しているので、プロパティ保存領域の更新と７セグ LED の表示変更を行えば済むように思いますが、設定コールバック関数が呼ばれるタスクはミドルウェアのタスクで、この関数を呼んだタスクはメインタスクですので、メモリに不整合が起こるかもしれません。排他処理もせず、設定コールバック関数をここから呼ぶのは避けてください。

自ノードであっても、電文による制御を行います。API「ecn_esv_setc」関数で動作状態プロパティ（0x80）を１バイトのデータで ON なら 0x30、OFF なら 0x31 の電文を作成します。API「ecn_snd_esv」を使用して電文を送信します。

ミドルウェアのタスクに電文が届くと、設定コールバック関数が呼ばれて動作状態プロパティが更新されます。また、状態変化時アナウンス属性を持っているので、外部へ「プロパティ通知」が送信されます。

（ア）　メインタスクとミドルウェアのシーケンス



プロパティの状態変化があった場合、通知電文送信のため取得コールバック関数が呼ばれます。

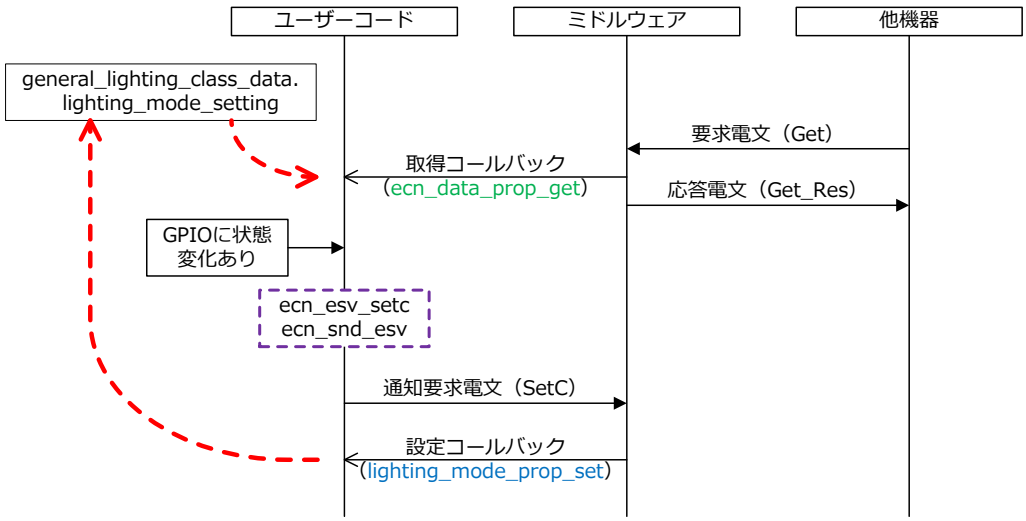
【４】　ボタン２状態変化処理

```
/*
 * ボタン２状態変化処理
 */
static void main_btn2_change(bool_t push)
{
	ER ret;
	T_EDATA *esv;
	uint8_t p_edt[1];

	/* 押されて戻った時に処理する */
	if(push)
		return;

	/* 点灯モードの切り替え */
	switch(main_mode){
	/* 自動の場合 */
	case lighting_mode_auto:
		/* 通常灯に変更 */
		main_mode = lighting_mode_normal;
		p_edt[0] = 0x42;
		break;
	/* 通常灯の場合 */
	case lighting_mode_normal:
		/* 常夜灯の場合 */
		main_mode = lighting_mode_night;
		p_edt[0] = 0x43;
		break;
	/* 常夜灯の場合 */
	case lighting_mode_night:
		/* カラー灯の場合 */
		main_mode = lighting_mode_coler;
		p_edt[0] = 0x45;
		break;
	/* カラー灯の場合 */
	case lighting_mode_coler:
	default:
		/* 自動の場合 */
		main_mode = lighting_mode_auto;
		p_edt[0] = 0x41;
		break;
	}
	/* プロパティ設定電文作成 */
	ret = ecn_esv_setc(&esv, GENERAL_LIGHTING_CLASS_EOBJ, 0xB6, 1, p_edt);
	if(ret != E_OK){
		syslog(LOG_ERROR, "ecn_esv_setc");
		return;
	}

	/* 電文送信 */
	ecn_snd_esv(esv);
}
```

ボタン２の処理は、点灯モードの切り替えですが、ON/OFF の２つの値が、自動／通常灯／常夜灯／カラー灯の４つの値になっただけで処理方法は変わりません。

ボタン１と同様に、プロパティ書き込み要求 API で電文を作成し、送信します。それをミドルウェアタスクに届いた時に、設定コールバック関数が呼ばれて、点灯モードの変更が行われます。

（ア）　メインタスクとミドルウェアのシーケンス



状態変化時アナウンス属性がないので、取得コールバックは呼ばれません。

1.6. ポートを直接アクセスするコールバック関数

プロパティの定義では、必ずしも前述のような保存領域が必要になるわけではありません。例として保存領域を持たないコールバックの定義を説明します。

【1】　現在時刻設定と現在年月日設定プロパティ

プロパティ名称	EPC	プロパティ内容 値域	データ型	データサイズ(Byte)	アクセスルール	必須	状態変化時アナウンス	
動作状態	0x80	ON／OFFの状態を示す。 0x30=起動中,0x31=未起動中	unsigned char	1	Set Get	○ ○	○	
点灯モード設定	0xB6	自動／通常灯／常夜灯／カラー灯 0x41=自動,0x42=通常灯,0x43=常夜灯,0x45=カラー灯	unsigned char	1	Set/Get			
設置場所	0x81	設置場所を示す。 0x41=異常発生有,0x42=異常発生無	unsigned char	1or17	Set/Get	○		
規格Version情報	0x82	対応するAPPENDIXのリリース番号を示す。 1バイト目：0x00固定(for future reserved) 2バイト目：0x00固定(for future reserved) 3バイト目：リリース順をASCIIで示す。 4バイト目：0x00固定(for future reserved)	unsigned char×4	4	Get	○		
異常発生状態	0x88	何らかの異常の発生状況を示す。 0x41=異常発生有,0x42=異常発生無	unsigned char	1	Get	○		
現在時刻設定	0x97	現在時刻HH:MM 0x00～0x17:0x00～0x3B(=0～23:0～59)	unsigned char× 2	2	Set/Get			
現在年月日設定	0x98	現在年月日YYYY:MM:DD 1～0x270F:1～0x0C:1～0x1F(=1～9999:1～12:1～31)	unsigned char× 4	4	Set/Get			

機器オブジェクトスーパークラスには、現在時刻設定と現在年月日設定というプロパティが規定されていますので、この 2 つのプロパティを一般照明に追加します。

```
/* 現在時刻設定 */
ECN_DEF_EPRP (GENERAL_LIGHTING_CLASS_EOBJ, { 0x97, EPC_RULE_SET | EPC_RULE_GET, 2, (intptr_t)NULL,
(EPRP_SETTER *)time_prop_set, (EPRP_GETTER *)time_prop_get });

/* 現在年月日設定 */
ECN_DEF_EPRP (GENERAL_LIGHTING_CLASS_EOBJ, { 0x98, EPC_RULE_SET | EPC_RULE_GET, 4, (intptr_t)NULL,
(EPRP_SETTER *)date_prop_set, (EPRP_GETTER *)date_prop_get });
```

cfg ファイルに上記の定義を追加します。第五引数の拡張情報は使いませんので NULL を設定します。第六引数には現在時刻設定または現在年月日設定の設定コールバック関数を設定し、第七引数の取得コールバックにもミドルウェアが提供する関数ではなく、ユーザー定義の関数を設定します。

この 2 つのプロパティは、保存領域を持たいな実装とするので、一般照明クラスの構造体に追加の変数はありません。

（ア）　取得コールバック関数

```
/*
 * 現在時刻取得関数
 */
int time_prop_get(const EPRPINIB *item, void *dst, int size)
{
	uint8_t *p_dst;

	if(size != 2)
		return 0;

	/* 読み出し終了確認 */
	while((sil_reb_mem(RC1CC2) & RC1CC2_RWST) != 0)
		tslp_tsk(1);

	/* カウンタ停止設定 */
	sil_wrb_mem(RC1CC2, sil_reb_mem(RC1CC2) | RC1CC2_RWAIT);

	/* カウンタのウエイト状態の確認 */
	while((sil_reb_mem(RC1CC2) & RC1CC2_RWST) == 0)
		tslp_tsk(1);

	/* 時刻設定 */
	p_dst = (uint8_t *)dst;
	*p_dst++ = sil_reb_mem(RC1HOUR);
	*p_dst++ = sil_reb_mem(RC1MIN);

	/* カウンタ動作設定 */
	sil_wrb_mem(RC1CC2, sil_reb_mem(RC1CC2) & ~RC1CC2_RWAIT);

	return (intptr_t)p_dst - (intptr_t)dst;
}
```

現在時刻設定の取得コールバックでは、CPU（V850ES/JH3-E）のリアルタイムカウンタから読み出した値を、ミドルウェアに返すように実装します。

```
/*
 * 現在年月日取得関数
 */
int date_prop_get(const EPRPINIB *item, void *dst, int size)
{
	uint8_t *p_dst;

	if(size != 4)
		return 0;

	/* 読み出し終了確認 */
	while((sil_reb_mem(RC1CC2) & RC1CC2_RWST) != 0)
		tslp_tsk(1);

	/* カウンタ停止設定 */
	sil_wrb_mem(RC1CC2, sil_reb_mem(RC1CC2) | RC1CC2_RWAIT);

	/* カウンタのウエイト状態の確認 */
	while((sil_reb_mem(RC1CC2) & RC1CC2_RWST) == 0)
		tslp_tsk(1);

	p_dst = (uint8_t *)dst;
	*p_dst++ = 0x20;
	*p_dst++ = sil_reb_mem(RC1YEAR);
	*p_dst++ = sil_reb_mem(RC1MONTH);
	*p_dst++ = sil_reb_mem(RC1DAY);

	/* カウンタ動作設定 */
	sil_wrb_mem(RC1CC2, sil_reb_mem(RC1CC2) & ~RC1CC2_RWAIT);

	return (intptr_t)p_dst - (intptr_t)dst;
}
```

現在年月日設定についても、同様にリアルタイムカウンタから読み出した値を返します。

このように、CPU のレジスタなどメモリ以外からプロパティを返すことは可能です。取得コールバック関数を定義することで、プロパティとは違った表現で保存されている値を、変換処理などを行ってミドルウェアに渡すことが出来るようになります。

（イ）　設定コールバック関数

```
/*
 * 現在時刻設定関数
 */
int time_prop_set(const EPRPINIB *item, const void *src, int size, bool_t *anno)
{
	uint8_t *p_src;

	if(size != 2)
		return 0;

	/* 書き込み終了確認 */
	while((sil_reb_mem(RC1CC2) & RC1CC2_RWST) != 0)
		tslp_tsk(1);

	/* カウンタ停止設定 */
	sil_wrb_mem(RC1CC2, sil_reb_mem(RC1CC2) | RC1CC2_RWAIT);

	/* カウンタのウエイト状態の確認 */
	while((sil_reb_mem(RC1CC2) & RC1CC2_RWST) == 0)
		tslp_tsk(1);

	/* 時刻設定 */
	p_src = (uint8_t *)src;
	sil_wrb_mem(RC1HOUR, *p_src++);
	sil_wrb_mem(RC1MIN, *p_src++);
	sil_wrb_mem(RC1SEC, 0x00);

	/* カウンタ動作設定 */
	sil_wrb_mem(RC1CC2, sil_reb_mem(RC1CC2) & ~RC1CC2_RWAIT);

	return (intptr_t)p_src - (intptr_t)src;
}
```

取得コールバックと同様、CPU のリアルタイムカウンタを直接操作するように実装します。

```
/*
 * 現在年月日設定関数
 */
int date_prop_set(const EPRPINIB *item, const void *src, int size, bool_t *anno)
{
	uint8_t *p_src;

	if(size != 4)
		return 0;

	/* 書き込み終了確認 */
	while((sil_reb_mem(RC1CC2) & RC1CC2_RWST) != 0)
		tslp_tsk(1);

	/* カウンタ停止設定 */
	sil_wrb_mem(RC1CC2, sil_reb_mem(RC1CC2) | RC1CC2_RWAIT);

	/* カウンタのウエイト状態の確認 */
	while((sil_reb_mem(RC1CC2) & RC1CC2_RWST) == 0)
		tslp_tsk(1);

	/* 年月日設定 */
	p_src = (uint8_t *)src;
	p_src++;  /* 20XX */
	sil_wrb_mem(RC1YEAR, *p_src++);
	sil_wrb_mem(RC1MONTH, *p_src++);
	sil_wrb_mem(RC1DAY, *p_src++);

	/* カウンタ動作設定 */
	sil_wrb_mem(RC1CC2, sil_reb_mem(RC1CC2) & ~RC1CC2_RWAIT);

	return (intptr_t)p_src - (intptr_t)src;
}
```

設定コールバック関数の場合は、引数で渡されるプロパティのサイズや値が正しいか確認するように実装します。