

Internet of Things Solution for Smart Farm (Tulip Plantation)



Kantri Yangsunit
Nitichai Kunapilukkul
Sutiwat Bunditthai

Bachelor of Science in
Engineering and Technology Management
Faculty of Engineering
King Mongkut's Institute of Technology Ladkrabang
Academic Year 2019

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้



COPYRIGHT 2019
FACULTY OF ENGINEERING
KING MONGKUTS INSTITUTE TECHNOLOGY LADKRABANG

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Acknowledgement

We would like to show our gratitude to our thesis advisor, Dr. Pimprapai Thainiam for her responsibility and advice that help us walkthrough the research. We at most appreciated for her advice that give to us in many ways. Moreover, she also gave us lots of life's methodologies that help us in the future. We might not finish this project completely if we did not get those advice and support from her.

In addition, we are fully grateful for the profession of Engineering and Technology Management faculty: Dr. Todsai Chumwatana, and all people that give suggestions and help.

Finally, I most thankfully to university and my friends for all their support during the time we do this project.

Abstract

The aim of this Engineering and Technology Management project is to create a development environment, so-called “IoT Smart farm”, for improving the traditional agriculture and help a farmer to solve the problems from a natural disaster; such as, insect invasion, flood, and bad weather conditions. Moreover, our Smart Farm also increases productivity by controlling the external factor that might cause a reduction in agricultural production.

Farmuto’s name comes from two words “Farm” and “Auto,” meaning farm that can control by itself. IoT Smart farm consists of four components: temperature control, humid control, light control, and water control. The temperature must be controlled to make the product more productive and have the consistency to survive in any location. Humid is one of the factors that must be controlled because each plant’s requirements are different. Light also have to control, because light is one of the factors that raise productivity and help to make a plant grow faster and stronger. Lastly, water is one of the most important factors that must be controlled strictly.

Farmuto’s system is developed by collecting data from the Arduino sensor about the environment in the greenhouse. The whole system includes a soil moisture sensor, temperature sensor, air system, water spray system, and mobile application. Arduino is also included in the system, in order to get all of the information from the sensor. C++ is our selected programing language used to create the application and to get the real-time sensors.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Table of Contents

Chapter 1 Introduction.....	1
1.1 Motivation.....	1
1.2 Objectives	1
1.3 Scope of Work	2
1.3.1 Plantation	2
1.3.2 Hardware.....	2
1.3.3 Application.....	2
1.4 Benefit.....	3
Chapter 2 Literature Review	4
2.1 Smart Farm DIY	4
2.2 Smart Farm (Thailand).....	5
2.3 Tulip Type and Background Information	6
2.4 Tulip's Plantation.....	8
2.5 Tulip's Plantation in Thailand	9
2.6 Smart Farm's Concept.....	10
2.6.1 Smart Farm's Concept.....	11
2.6.2 Features in Smart Farm System	11
2.6.3 Adaptive policies for Smart Farm.....	13
2.6.4 Succeeded Factor to Achieve in Smart Farm Policy	16
2.6.5 Monitoring and evaluation.....	19
Chapter 3 Research Methodology	20
3.1 Process of Work.....	21
3.1.1 Tulip.....	21
3.1.2 Prototype Box	22

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

3.1.3 LED Light	24
3.1.4 Water Pump	27
3.2 Stage of Growth of Each Weeks	28
3.3 Application.....	31
3.4 Internet of Things.....	31
3.5 Application User Interface	32
3.6 Application User Experience	37
3.7 Hardware System and Programming	38
3.7.1 Hardware.....	38
3.7.2 Arduino's Work Example	40
3.7.3 Software	42
Chapter 4 Experimental Results.....	44
4.1.1 Greenhouse Result	44
4.1.2 Light and Water Result	45
4.1.3 Application Result	45
4.1.4 Arduino Board Result	45
4.2 Use Case Diagram.....	47
4.2.1 Brief Use Case	47
Chapter 5 Conclusions and Discussions.....	49
References.....	51
Appendix A.....	55

List of Figures

Figure 2.1 This is a figure of Smart Farm DIY website	5
Figure 2.2 This is a figure of Smart Farm (Thailand) website.....	6
Figure 2.3 This is a figure of Foxtrot Tulips.....	7
Figure 2.4 This is a figure of Variety of Tulip Breed	8
Figure 2.5 This is a figure of Refrigerator Plantation.....	10
Figure 3.1 This is a figure of Tulip at the First Stage	21
Figure 3.2 This is a figure of Prototype box	23
Figure 3.3 This is a figure of Power Supply	23
Figure 3.4 This is a figure of After Take Out Form the Fridge	24
Figure 3.5 This is a figure of Light Spectrum Length	25
Figure 3.6 This is a figure of Light	26
Figure 3.7 This is a figure of Water Pump.....	27
Figure 3.8 This is a figure of Week 1 st	29
Figure 3.9 This is a figure of Week 3 rd	29
Figure 3.10 This is a figure of Week 6 th	30
Figure 3.11 This is a figure of Rotten Tulip at Week 4 th	30
Figure 3.12 This is a figure of Smart Farm IOT Adaptivity	32
Figure 3.13 This is a figure of Farmuto Application's Login Page	33
Figure 3.14 This is a figure of Farmuto Application's Controller Page	34
Figure 3.15 This is a figure of Farmuto Application's Graphical Dashboard Page	35

Figure 3.16 This is a figure of Farmuto Application's Timer Page.....	36
Figure 3.17 This is a figure of Farmuto Application's Setting Page	37
Figure 3.18 This is a figure of the Automatic Circuit.....	40
Figure 3.19 This is a figure of Soil Moisture Sensor Module	40
Figure 3.20 This is a figure of Water Flow Sensor.....	41
Figure 3.21 This is a figure of NodeMCU for ESP8266	41
Figure 3.22 This is a figure of XCode	42
Figure 3.23 This is a figure of Firebase Database	43
Figure 3.24 This is a figure of Arduino IDE.....	43
Figure 4.1 This is a figure of Greenhouse Box Complete Form.....	45
Figure 4.2 This is a figure of Complete Application	46
Figure 4.3 This is a figure of Use Case Diagram.....	47

List of Tables

Table 3.1 The table of Light Spec.....26

Table 3.2 The table of Devices used in this project.....38



เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Chapter 1

Introduction

The project's idea comes from the problems that occur in the current situation, such as the insect's invader and flood that cause damage to the farmer product. The suffering of a farmer is the motivation that drives the project to develop the IoT system that will solve the problems.

1.1 Motivation

Currently, the way business operate are most likely using IOT to run in the business, which can increase productivity and efficiency of the product as well. In addition, Thailand is most likely becoming aging society and the new generation of population will be lacking skilled labor; so IOT smart farm will play an important role in the future. Moreover, Thailand is one of the countries that are facing with the labor shortage problem, this smart system will be considered as a significant solution to the agriculture in the future. which can reduce the cost of the agriculture in the long run as well. Therefore, we aim to increase the quality of life for farmers in both, management part and farming part.

1.2 Objectives

1. To develop an IoT system using for agriculture.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต่อ1อ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

2. To develop an application used to control all devices in the IoT system.

1.3 Scope of Work

The scopes of this project are listed as follows:

1.3.1 Plantation:

- Tulip is selected as our experimental subject because tulip is one of the toughest plants to plant in the warm temperature area, also tulip need a special care to keep it alive and make flower look beautiful.
- Plant tulip in the greenhouse box to qualify the quality of the project.

1.3.2 Hardware:

- Middle size plastic wrap greenhouse for prototype spec. height 1.4-meter × width 1.2-meter width. It is modified to add a temperature controller, humid controller, LED light, and water springer.
- All the devices must be connected to the Arduino panel that will send the information to mobile phone application.

1.3.3 Application:

- Application can show the real-time information of the temperature, humid, and soil quality via mobile phone.
- Application can control the system in the greenhouse by internet, such as, turning on or off the light.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

1.4 Benefit

There are several beneficial effects that will occur in the project. Firstly, it can control and improve the productivity of the product. Secondly, the greenhouse will control the quality of the product to get the same quality in every product. Moreover, the greenhouse can control the air temperature, so it can produce any seasonal plant in every season of the year. According to the close environmental greenhouse, it also prevents insects and weed that can damage the plant. Lastly, this project will benefit farmers as well. The farmer can control every part of the farm via mobile phones, so it will increase the quality of life and time for farmers.



Chapter 2

Literature Review

Smart Farm is one of the popular activities nowadays, using similar existing technology. Most of the companies provide the hardware and smart farm knowledge to the customer. There is two big Smart Farm company in Thailand that provide the service for people who interest in the smart farm, Smart Farm DIY and Smart Farm Thailand.

2.1 Smart Farm DIY

Smart Farm DIY is one of the companies that provide service online mostly aim at the mobile application that can control every component on the Smart Farm. Also, the company selling compact hardware that can use right after buying it. The outstanding point of this website is the customer does not need any installation part as shown in figure 2.1.

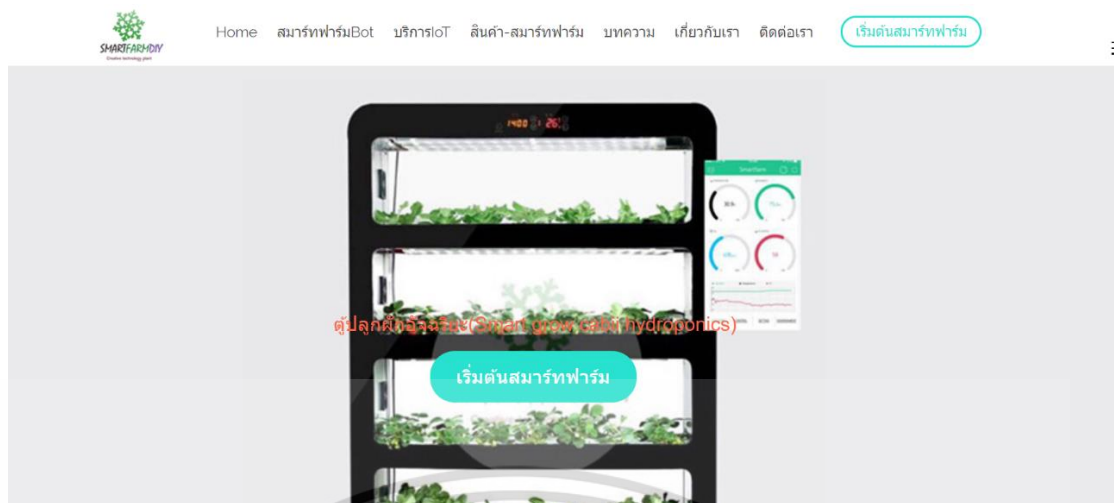


Figure 2.1 This is a figure of Smart Farm DIY website

Source: <http://www.smartfarmdiy.com/>

2.2 Smart Farm (Thailand)

Smart Farm (Thailand) mostly displays knowledge along with the product (hardware) they sell. The company aims to sell the hardware more than the service and application. Their website contains a lot of articles that help customers to understand the Smart Farm as shown in figure 2.2.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้



Figure 2.2 This is a figure of Smart Farm (Thailand) website

Source: <http://smartfarmthailand.com/>

2.3 Tulip Type and Background Information

Tulip is the plant in the Liliaceae family, that have originated from Southern Europe, North Africa, Asia (Anatolia and Iran), and East and Northern China. Most people understand that Tulip is originally from Holland or Netherland. In Fact, it not a flower that has its origin in Holland as the research says. In the 16th century, Dutch ambassadors in Turkey brought Tulip flower for Europe for the first time. Since then Tulip is become one of the important businesses in the country, because the weather, height, and land are suitable for the plant as shown in figure 2.3.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้



Figure 2.3 This is a figure of Foxtrot Tulips

Source: <https://www.gardenia.net/plant/tulipa-foxtrot-double-early-tulip>

Tulip is a monocotyledon plant that has sharp and small leaves, pointed at to the tip of the branches. Each leaf will grow in the opposite direction, which each plant will have around 3-4 leaves. Normally, tulips have a height between 12-18 inches, which depends on the type of tulip. The flowers of tulips are the same in any shape, design, and size that cup in the cup shape. However, each type of tulip's flower comes with a different color. Tulips have a unique signature that the flower will not bloom fully but will bloom by opening a little petal instead. In addition, the tulip has 2 layers of petal and each petal has 3 petals inside. Tulip has come with many different shades of color; red, orange, dark yellow, yellow, light yellow, pink, and white and many alternative colors as shown in figure 2.4.



Figure 2.4 This is a figure of Variety of Tulip Breed

Source: <http://strategycoaching.co/types-of-tulips/>

2.4 Plantation

Tulip can be planted only in the specific area, in cold weather. As usual, tulips will grow well in the temperature around 15-20 degrees Celsius, which suitable for plants in the Winter. In this shoot, there are several layers of thick, moist membranes overlapping. These layers cover the strong stems. Botanically, the collector called them bulb.

Planting tulips are more different from other florets because the tulip bulb will not grow normally in this weather. The bulb must be cover in the paper bag or plastic bag with a little hole on it and put in the fridge for around 4-6 weeks (up to 8 weeks if

possible). The infusion of the bulb will help speed up flowering because during the เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

infusion the bulb will convert carbohydrate in the bulb into sugar which will be stimulating growth the head. This method will help the flower bud to open before coming out to experience the hot weather outside. Moreover, it will help the flower to bloom longer. After takeout off the fridge, bring tulip head to the ground around 4 inches. Tulip will take 39 days to bloom and after they bloom, they can last for only 7 to 15 days.

Watering tulips are more dedicated than another floret because tulip needs lots of water plus those water must be a little cold. For the first week after the head on the ground, the soil must be wet all the time. After the leave grows up ahead the ground, tulip still needs water every day and twice a day. Tulip's bulbs are sensitive, if it does not get enough water it may survive and if it gets too much water it will die due to the mold.

2.5 Tulip Plantation in Thailand

According to the weather restriction in Thailand that has hot air, but tulip sensitively needs only cold air. Department of Agriculture Extension at Doi Inthanon, Chiang Rai have researched on how to plant tulip flower in Northern Thailand. Chiang Rai Agricultural Occupation Promotion and Development Center (Highland Agricultural Extension) and Agricultural Occupation Promotion section 6 have found that there are about 20 types that can plant in Thailand. Only 6 types that had plants which are Christmas Sweet (white), LLC de France (red), Alaska (pink), Leen Vd Nark (velvet red), Negrita (purple), and Strong gold (yellow gold).

Central Thailand, Bangkok, have a quiet hot temperature that tulip may not survive or grow perfectly. To plant tulip in Bangkok, the greenhouse is important to have a 60% shading net. The greenhouse must control the weather around 15-25 degrees Celsius. The greenhouse must contain at least a fan that works during the mist sprayer work and the sprayer must spray water twice a day. As you can see the difficulty of planting tulip in the city, so they are another trend that plant tulip in the refrigerator as shown in figure 2.5.



Figure 2.5 This is a figure of Refrigerator Plantation

Source: <https://www.bloggang.com/mainblog.php?id=moving-moon>

2.6 Smart Farm's Concept

1. Specification of Smart Farm System
2. Features in Smart Farm System

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

3. Adaptive policies for Smart Farm
4. Success factors
5. Monitoring and evaluation

2.6.1. Smart Farm's Concept

Smart Farming is the introduction to modern technology that consists of computer systems, telecommunications, sensor systems, and biotechnology that work with agricultural and Geoengineer. It will bring modern technology together, such as changing the ground that cannot plant at all. For example, desert areas, to become food sources in the future, which Smart Farm will be occurred more in the future of Thailand and occurring more around the world. Moreover, Smart Farm will be more complete with various technologies that have developed rapidly combined with the internet infrastructure. In addition, governments start to focus on agriculture by providing equipment and knowledge such as the weather check tool kit, soil's humid sensor, PH sensor, and Biotechnology that will handle farmer's problems in the future.

2.6.2 Features in Smart Farm System

All technology will change an ordinary farm into a smart farm (Intelligent Farm). The current Smart Farm is a semi-automatic. When the appropriate technology is allocated, it can be managed such as controlling the water usage system and automatic light control, which is currently being used

but still having a high price and not being proper use. To use the technology correctly, there is 3 basic idea:

1) Use the various sensors to track the data and activities on the farm, including the use of mechanical robots and intelligent devices on the farm.

2) Use mobile devices to get involved in the farm and to access various data from anywhere.

3) Use cloud computing to assist in processing, cultivating, and bringing various factors from the farm together. Cloud computing can store production data and product conditions from the farm and link to all services in the Value Chain. The value chain consists of products in good condition and has an attractive quality. When all this factor is linked to the cloud and other variables that can be calculated mathematically. A farmer can use the data collected to write an algorithm that can be classified to specify each step of work and able to anticipate future directions of work. Farmers can also analyze the product prices from the market by using cloud data. Currently, many countries have adopted this method, including China, Japan, Korea, Singapore, America, and Australia.

2.6.3 Adaptive policies for Smart Farm

According to the Ministry of Agriculture and Cooperatives have provided a guide to driving Smart Farmer policies. They mention the concept and guidelines for Smart Farm implementation with details as follows (Ministry of Agriculture and Cooperatives,2561: Online)

Smart Farm concept is one of the 4 strategic issues of that Ministry of Agriculture and Cooperative plan to develop the Thailand agriculture field, which goes along with the 11th National Economic and Social Development Plan.

In order to develop farmers to be self-reliant and being immune to the risks that might happen all the time in the national market. Moreover, to produce and marketing capabilities at a level that is ready to become a manager of the professional farms that have successfully cultivated agriculture.

Agriculture in Thailand still faces many problems. Especially, productivity is still not high and the average farmer's income is low because farmers do not have knowledge of marketing and production planning. In addition, a farmer is still lack of agriculture's knowledge to produce a high-quality product that is safe for customer and environment. Nowadays, farmers still lack sustainable development. Especially the strengthening of self-sufficiency in the long-term, which will make the government do not need to Establishing policies for solving immediate problems and embracing them. The Smart Farm concept is therefore an important mechanism to respond to the

เอกสารนี้เป็นเอกสาร **farmer**. นวัตกรรมสำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และดัดแปลงอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

From the point of view of the National Electronics and Computer Technology Center, which presents Smart Farm Flagship in agriculture, the main concept of Smart Farm is to use electronic technology and computers(IoT) Including information technology in the development of the entire supply chain Agricultural products to upgrade Product standardization and cost reduction through agricultural development in 4 areas which are 1) reducing costs 2) increasing production quality and product standards 3) reduce risks from pests and natural disasters and 4) knowledge management. As for the Smart Farmer concept from the Ministry of Agriculture's perspective referring to farmers who have accurate knowledge of agricultural occupations who can make plans by knowing the market demand and prepare production in accordance Including the ability to increase production efficiency and solve problems fast enough by analyzing all-round information as a decision-making component based on principles and reasoning as well as knowing the application of technology that can adapt to use in agriculture field.

In terms of productivity, Smart Farm is focusing on produce high quality, safe, and environmentally friendly. In terms of income, farmers will have more income than before at least at the same level or higher than the minimum wage, which means the income will increase on average no less than 180,000 baht or equal to the minimum salary Bachelor's degree.

Important operations are an establishment of the War Room (Agriculture Data) to connect and cover all data from each sector. Another aspect of creating Smart Farmer is the development of "Smart Officer" or

government officials who have Academic knowledge and policy that can apply technology to support farmers efficiently.

According to the Smart Farm concept, if successful, it will benefit the Ministry of Commerce's operations by reducing costs and increasing the quality of production standards that will ease the burden of the Ministry of Commerce on overseeing agricultural prices that are always volatile. This mechanism will make it less necessary to implement product price subsidy policies and able to promote more market mechanisms by itself. Therefore, the Ministry of Commerce should support the Ministry of Agriculture by allow Smart Farmer to connect and link to marketing data with production data. Especially the integration of databases with agricultural data centers for in-depth marketing, which will enable Smart Farmer to have access to in-depth market data for production planning and pushing this idea to create concrete change for the agricultural sector.

Smart Farmer concept from the perspective of the Ministry of Agriculture Refers to farmers who have a thorough knowledge of agriculture and excellent planning idea. Being aware of market demand and preparing production to meet demand Including the ability to increase production efficiency and readiness to step into the ASEAN Economic Community.

To develop the farmers to become Smart Farmer, the Committee for Smart Farmer Policy and Smart Office of the Ministry of Agriculture and Cooperatives has undertaken the following actions:

- 1) Define the characteristics of a smart farmer by screening and select smart farmer to classify farmers into 3 main groups, consisting of 1) smart farmer prototype 2) existing smart farmer and 3) developing smart farmer
- 2) Learn from a prototype that already exists as a body of knowledge and a suitable model to other farmers, including officials and public who are interested.
- 3) Defining projects and activities for farmer development to be a smart farmer with qualifications as specified.
- 4) Extension of agricultural career development throughout the chain of agricultural products.
- 5) Monitoring and evaluation of Smart Farmer development according to smart farmer policy.

2.6.4 Succeeded Factor to Achieve in Smart Farm Policy

- Cooperation of internal and external agencies in the Ministry of Agriculture and Cooperatives.
 - The composition of the committee that has representatives from internal and external agencies of the Ministry of Agriculture and Cooperatives must understand and be aware the importance of the roles and responsibilities of each department/organization in driving policy, including collaboration between the two

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

countries and between central and provincial areas to integrate work efficiently.

- Support from the central division of the Ministry of Agriculture and Cooperatives found that the initial stage is the major problem. Most of the problems are from many areas such as insufficient in people, insufficient budget, and some department did not receive cooperation from other departments as it should be. The operations in the area are quite limited and many departments in the area have many works to do at the same time. Centralized operations are not as good as they should be, which is a common problem. In the future, the Ministry of Agriculture and Cooperatives must support more into the provincial level. Including the central departments must support them by providing more knowledge, budgets, and operational tools.
 - Define the mechanism of work to make it clear and without duplication, especially the appointment of a committee/working group. The task should appoint to a single committee or proceed by using only one set to save time and resources in management.
 - Use an efficient monitoring and evaluation mechanism with Smart Farmer and Smart Officer policy. The project must establish mechanisms and methods to monitor and report the results periodically to ensure that operations are conducted. The goal is to monitor results to assess the quality and appropriateness of methods and mechanisms that have been implemented.

- Project's Knowledge Development Goal

Knowledge management is intended to support the knowledge workers to be more efficient. It is a mechanism that caused them to gather more knowledge that is scattered everywhere to come together in one place. Knowledge storage is the way to gather people together in order to exchange their knowledge.

Knowledge can be divided into two major categories. First, explicit knowledge, which is written in a literal way such as manuals, books, websites, blogs, and hidden knowledge. Second, tacit knowledge, which is hidden in the person that experts in each area they are working on. Useful knowledge in the operation level can be divided into 4 levels, which are 1) “know-what” is the knowledge of facts and clear knowledge that they gained from the study, 2) “know-how” is the knowledge that has the characteristics of linking it to any case scenario. 3) “know-why” is the knowledge that based on reasoning in complex matters, and 4) “care-why” is the level of creativity and innovation in the workplace.

The main reason of knowledge management is the use of knowledge to increase efficiency and effectiveness in operations, which have 3 mainly goals 1) to develop work to have better quality and better results, 2) to develop people or workers in the organization, and 3) to develop the corporation's problems.

2.6.5 Monitoring and evaluation

The Committee has set guidelines for monitoring, reporting, and evaluating the result to Smart Farmer as follows.

- 1) Gather the local operational results for the Ministry of Agriculture to evaluate the qualities in each group according to the responsible branch or the department's mission and report the summary to the committee.
- 2) Collect and evaluate the development of farmers according to the qualifications in each branch that the department is responsible for and report problems back to the committee, so the project can improve.
- 3) Monitoring and evaluation of the overall policy of Smart Farmer

Chapter 3

Research Methodology

According to the research, Smart Farm is the project that suits the big farm and middle-size farm. Because many systems in the processes use lots of budgets to obtain a “Smart” greenhouse.

Fund or investment, due to the cost that quite high at the beginning and maintenance cost that farmers must pay all along that will make farmers not capable to pay. However, if the farmer knows how to adapt into the future by using new technology to use such as sensor and cloud computing, the business will gain more reliability and have more opportunity to grow faster and stronger than other.

Material:

- 1) Green House for prototype spec. height 1.4-meter × width 1.2-meter width. Can plant approximately 10 tulips. Each tulip needs around 15cm × 15cm space. The Cost of the prototype greenhouse box is 8,000 Baht.
- 2) Seed tray, pot, coconut blend, soil, and fertilizer
- 3) Tulip bulbs “Foxtrot”; 10 seeds are plant in the greenhouse and 6 are plant outside the greenhouse.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

3.1 Process of Work

3.1.1 Tulip

Firstly, order tulip bulbs from an online seller (import from Holland) and put one of each in the plastic bag and put it all in the refrigerator for 6 weeks. The result is some of them grew up a little bit and some leaves grew out from the bulb. However, the problem of putting tulip's bulb into the fridge is humidity in the fridge. The humidity is the most thing that damages the bulb because humid makes mold come into the bulb and damage some of the seed as shown in figure 3.1.



Figure 3.1 This is a figure of Tulip at the First Stage

The second process, after taking the bulb out of the fridge, we put the tulip's bulb separately into each pot to make it grew the roots before putting all the bulbs into the ground. In addition, we put 10 pots into the prototype greenhouse and 7 outside the prototype greenhouse. The reason for planting it separately because we want to specify the difference between planting Tulip in the control temperature with planting tulip at room temperature. Each tulip is getting the same method to grow, which includes the same amount of water twice a day and the same amount of fertilizer (8 seed of chemical fertilization). The only difference between these experimental is temperature that the tulip gets from the light (sun).

3.1.2 Prototype Box

The prototype was made of all metal in order to handle the weight of soil and cover every side of the box with 40mm PVC clear plastic to keep the air inside, which makes the box easy to control the temperature. At the bottom of the box had made to be a metal grille and put another sackcloth on top of the grille, so after putting the soil and water it will not spill on the ground, and water can go through the sack easily. All sides of the plastic are seal with silicone, so the air will not leak out. After the silicone dry, the system is a priority in order to make this project work as shown in figure 3.2. Firstly, start with the power supply that will be the main energy source because every sensor is using 12w electricity. In order to convert from home electricity at 220w to 12w, power supply will be the main source for the project as shown in figure 3.3. Secondly, set the LED light at the top of the box 1 foot long above the plant. Next are set

the water system in the middle of the box. Lastly, put a small air condition on the right of the box.

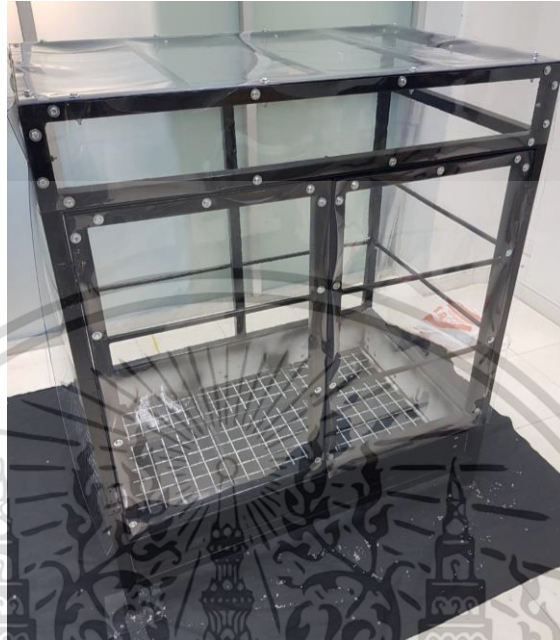


Figure 3.2 This is a figure of Prototype box



Figure 3.3 This is a figure of Power Supply

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

When the priority systems in the box are completed, we put in soil and move tulip in the box as shown in figure 3.4, so the project can keep tracking the result of the plant. The second priority is moving in the box, all Arduino sensors are set in the soil, air, and temperature control. Each of the sensors has their specific job, which will make the result more accurate.



Figure 3.4 This is a figure of After Take Out Form the Fridge

3.1.3 LED light

LED light that we use is specifically used in order to grow the plant as shown in figure 3.5, due to the wavelength and color that perfectly close to the sun and have the frequency of light that plants want the most. When the plant gets correct wavelength of light, it will make the photosynthesis process happen which makes the plant grow faster. Natural sunlight includes a light spectrum that has a wavelength between 200-5000 nanometer. However, plants just want only 380-770 nanometer which LED light what we use are specifically made

for plants. Other benefits are not as hot as the sun, so it was not enough to

damage the plant and flower. LED light include red and blue color, red color is made specifically to the synthesis of chlorophyll A&B, best for plants blooming and fruiting, and blue color for promoting the synthesis of chlorophyll, carotenoids, boosting photosynthesis to promote the left growth. Light spec as shown in, Table 3.1 and example of light as shown in figure 3.6.

Wavelength (nm)		Effect on Plant Growth
UV (ultraviolet)	280	Significantly reduces quantum yield and rate of photosynthesis.
	315~400	Promotes pigmentation, thickens plant leaves, and may be used to prevent harmful insects.
Visible Spectrum	440~470	Chlorophyll absorption peaks at 439nm and 469nm. The blue spectrum is the most efficiently absorbed spectrum, promoting mainly vegetative growth.
	510	Quantum absorption in the green spectrum. Little absorption is the yellow spectrum.
	610	No chlorophyll benefit. Efficiently absorbed by algae phycoerythrin and phycocyanin receptors.
	640~660	Chlorophyll absorption peaks at 642nm and 667nm. 660nm is the most vital wavelength for flowering. Speeds up seed germination and flower/bed onset.
	740	Emerson Enhancement Effect - quantum yield of red light and far red light, when shone simultaneously on a plant, increases the rate of photosynthesis.
Infrared	1000~1400	No plant activity detected at this wavelength. Heat generated.

Figure 3.5 This is a figure of Light Spectrum Length

source: <https://www.urbanvine.co/blog/11-core-guidelines-urban-farmers-must-know-before-buying-grow-lights>

Table 3.1 The table of Light Spec

1000W grow light	
Light Type	1pcs Led grow light
Nominal power	1000W
Actual power	200W
LED size	SMD 2835
LED quantity	2009pcsLEDS
LED color	1715pcs Red + 294pcs Blue
Lumen	3500LM
Beam angle	140°
Irradiation area	1~3 sq. meters
Voltage	AC85-265V
Size	310mm*310mm*35mm



Figure 3.6 This is a figure of Light

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และตั้ง 26 ข้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

3.1.4 Water Pump

The water system in the greenhouse is using a small size pump, 12v, 4.8 bar, for pumping water from the water storage beside the greenhouse into the greenhouse which can spray about 3.5 liters/minute or 210 liters/hour which is enough for tulip and to wet the soil. The pump also comes with 12v for electric, however, it comes with its own power supply that converts the current into 220v home electrical power as shown in figure 3.7. DC12V Model: Green-01 is specifically for mist sprayers so it will not damage the plant. In the greenhouse, we run for 3 mist springers to make it cover all the areas especially near the corner that may not get enough water. Each of the springers can cover for 2 feet and have 2 feet far from each other which covers the entire greenhouse.



Figure 3.7 This is a figure of Water Pump

3.2 Stage of Growth of Each Weeks

Tulip is the plant that needs quite a lot of water and quite often, so each time watering tulip we use about 21 liters or about 7 minutes which makes the soil wet enough for the tulip. However, mist sprayers will make the humidity too high because we spray a lot of water, so we add 2 mini fans that will make the airflow through the box and make the humidity flow out of the box. Humidity can be very dangerous to tulip because as the research says that might cause mold at the root and make the root rot. According to the size of prototype greenhouse, watering tulip twice a day might be too much because of the humidity, but in the bigger area up to 10x10 meter are needed to water tulip twice a day and this is one week for tulip as shown in figure 3.8

There are various big differences we found after three weeks as shown in figure 3.9, tulip that we planted in the prototype are growing taller and greener than the room temperature one, due to the differentiation of light and temperature. Moreover, the outsider tulip is having some issues with a big dark dot on the leave. So, we cut it off before it spread to other pots.

In the sixth week as shown in figure 3.10, the plant was moved from the pot into the ground in order to grow their root widely in a bigger area. From 16 pots, 6 outsider plants died after week fourth as shown in figure 3.12 and the other 10 are survive for 8 pots. After taking the plant out from the pot, we had checked the problem and the root. Two of the bulbs are died because of the mold and get too much water. After that, due to the Coronavirus that spread all around our project are not capable to continue so the project more like to control for long distance. More than two weeks that cannot continue doing the project.



Figure 3.8 This is a figure of Week 1



Figure 3.9 This is a figure of Week 3

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้



Figure 3.10 This is a figure of Week 6



Figure 3.11 This is a figure of Rotten Tulip at Week 4

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

3.3 Application

Application is one of the best options for controlling all the sensors in the smart farm area because it can control from everywhere in the network area. Our main control board is completely on the application. The most important of the application are UX and UI that must be simple and easy for users to use. This application call Farmuto, that will support both Android operation system and Apple operation system. Farmuto's Application contains two main parts: graphical dashboard and controller.

Graphical dashboard that will show various information that was captured from the sensor and send it back in real-time. On the graphical dashboard, the application will show the temperature in the greenhouse, humidity in the air, humidity in the soil, and water spend.

The controller will contain a water timer that can change the user can set more time for the soil's humidity is lower than usual. Also, the light timer can set on the user own time.

3.4 Internet of Things

Internet of Things or IoT is the connection of objects in the environmental area that can communicate with each other via the wireless and wire Lan protocol. Each of the “things”, also known as “sensor”, have their job to work in their specific area. For example, soil's humidity sensor that can work only in soil. Also, every sensor can communicate with other things by the Internet technology that connects each of the value which make farmer have more convenient, cost saver, and time saver. Moreover,

farmer can use the data that have been collected by sensor to do the result analytic. Farmers can also control entirely farm by only on smartphones. IoT is adaptive and convenient that can change into many uses; For example, as shown in figure 3.12.

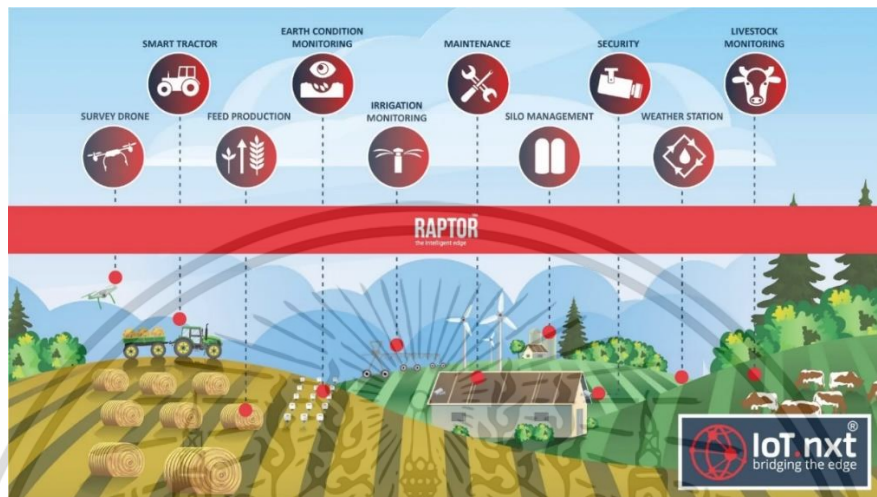


Figure 3.12 This is a figure of Smart Farm IOT Adaptivity
source:

3.5 Application User Interface

User Interface or UI is part of the application that contacts directly to the user such as button and surface platform. The main idea of UI is to focus on the design of the application that must be nice and easy for the user to understand. UI of the application must be user friendly which is the main idea that Farmuto tries to make the application. The Farmuto's application consists of 5 main pages which are Login page, Controller, Graphical dashboard, Timer, and Setting.

Login page is the most important part because to separate each sensor that has its own ID. Moreover, it will collect data from users that Log-in into Firebase Database, so we can use those data to separate users into groups for future use as shown in figure 3.13.



Figure 3.13 This is a figure of Farmuto Application's Login Page

Controller page will show the status of all sensors that connect to each account. This interface can be custom by add or delete sensors that users need or do not needed as shown in figure 3.14.

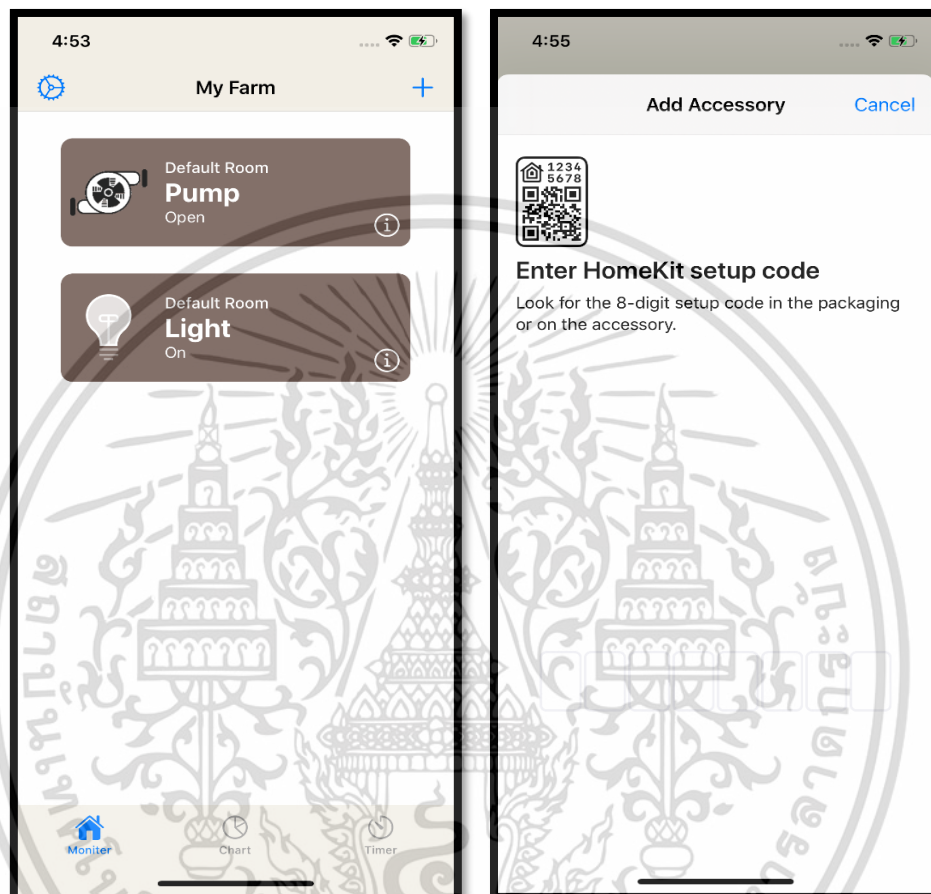


Figure 3.14 This is a figure of Farmuto Application's Controller Page

Graphical dashboard will show the collected data into a graph, so it makes users easy to make any decision as shown in figure 3.15.

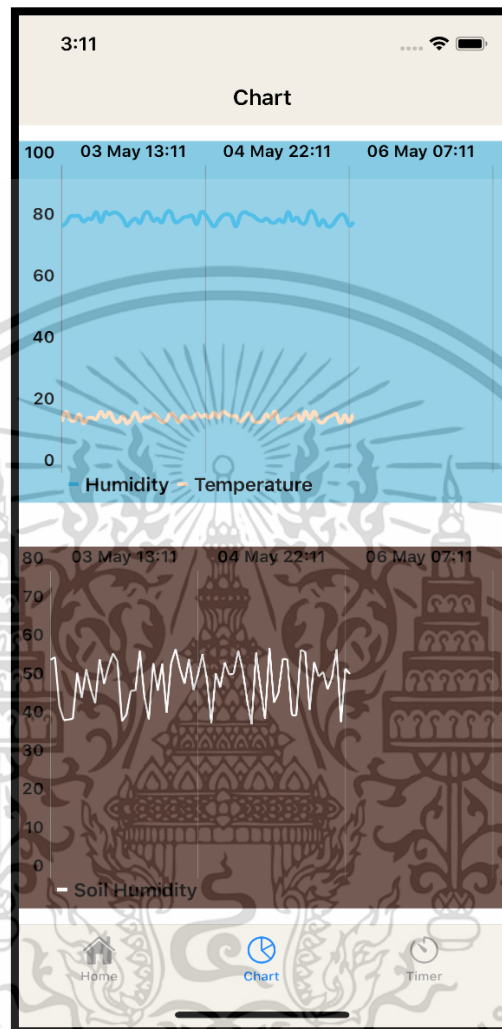


Figure 3.15 This is a figure of Farmuto Application's Graphical Dashboard Page

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Timer page will be the page that has a time that can set for special devices in the greenhouse, such as light and water springer in order user want to control the system manually. Time can be manipulated in time period and specific time event as shown in figure 3.16.

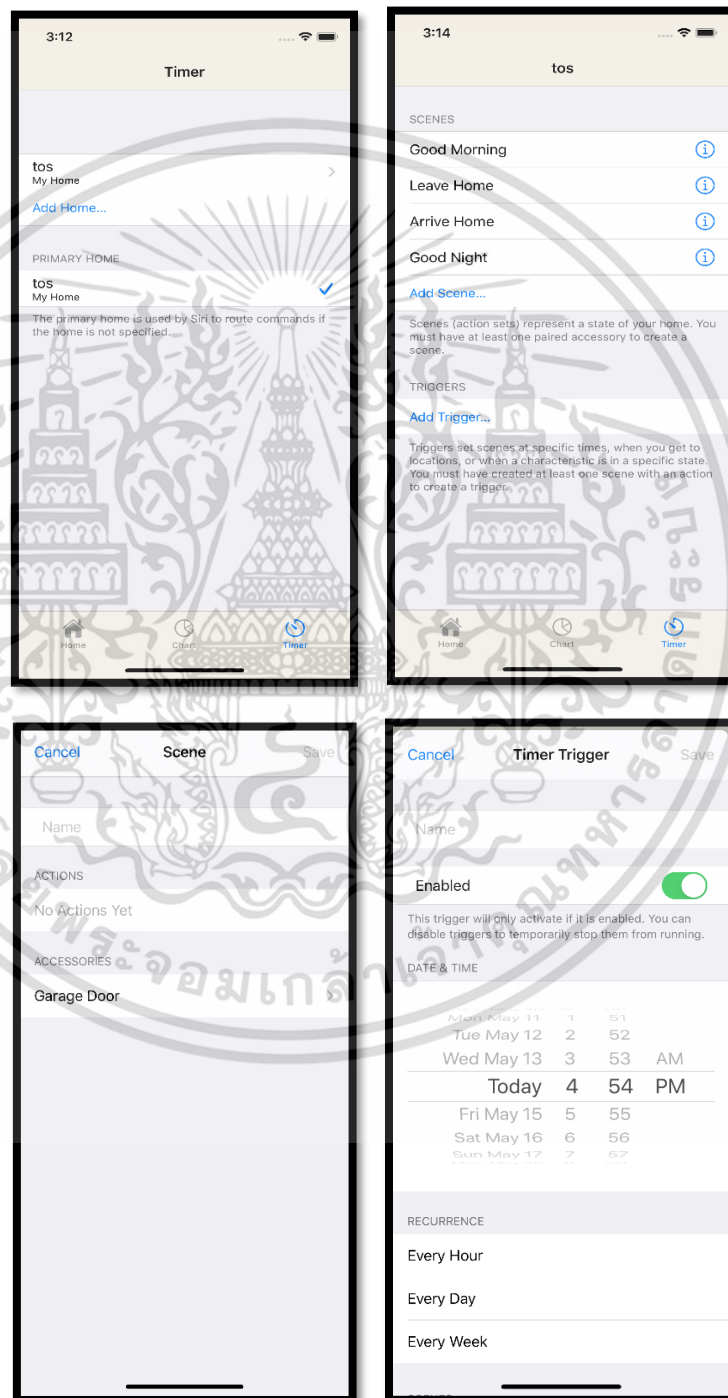


Figure 3.16 This is a figure of Farmuto Application's Timer Page

Setting will use for profile setting and log out as shown in figure 3.17.

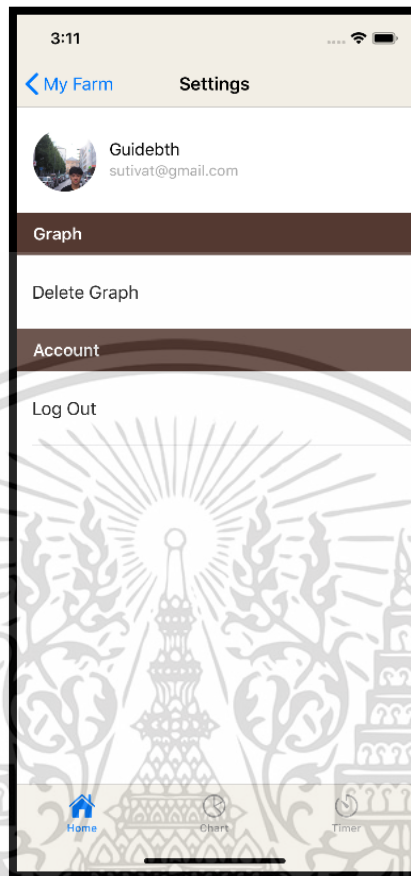


Figure 3.17 This is a figure of Farmuto Application's Setting Page

3.6 Application User Experience

User Experience or UX is an experience that the user will get after using the application. User Experience are including every part of the application that users can access. Farmuto's application are designs for a user to feel calm and reliable, so they can feel the simplicity of life.

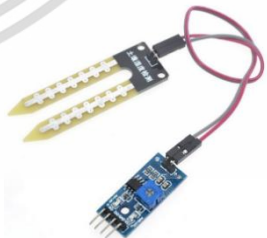
3.7 Hardware System and Programming

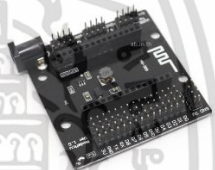
Farmuto's system can separate into 2 parts; hardware and software that work together.

3.7.1 Hardware

Hardware that we use for the control system are digital temperature, humidity sensor DHT11, soil moisture sensor module, water flow sensor, NodeMCU V3, power module, NodeMCU Base, and ESP01 relay module shown in table 3.2.

Table 3.2 The table of Devices used in this project

Name	Amount	Image
Digital Temperature and Humidity Sensor DHT11	1	
Soil Moisture Sensor Module	1	
Water Flow Sensor Flowmeter Hall Water Control 1-30L/min 2.0MPa	1	

NodeMCU V3 C340 Lua WIFI ESP8266-12E	3	
HLK-PM01 Power Module 5V 600ma	2	
NodeMCU Base Ver.1.0 for ESP8266	3	
ESP01 Relay Module for ESP8266	2	

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

3.7.2 Arduino's Work Example

Figure 3.18 - 3.21 is combine of devices for ready to use as an electrical controller

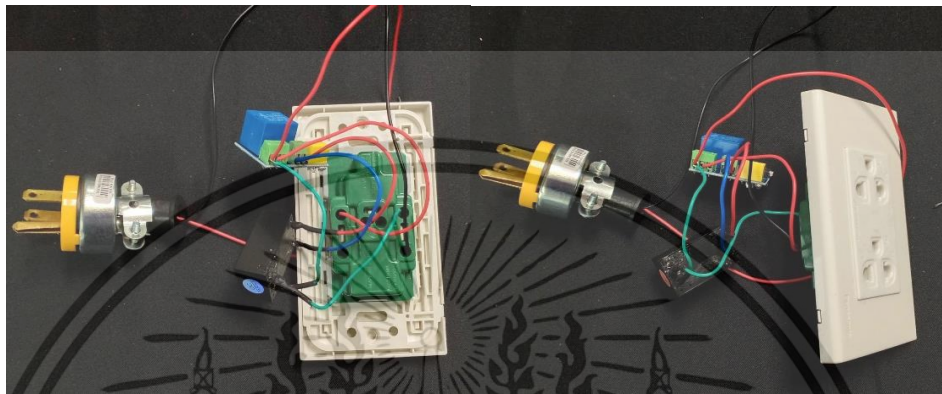


Figure 3.18 This is a figure of the Automatic Circuit



Figure 3.19 This is a figure of Soil Moisture Sensor Module

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

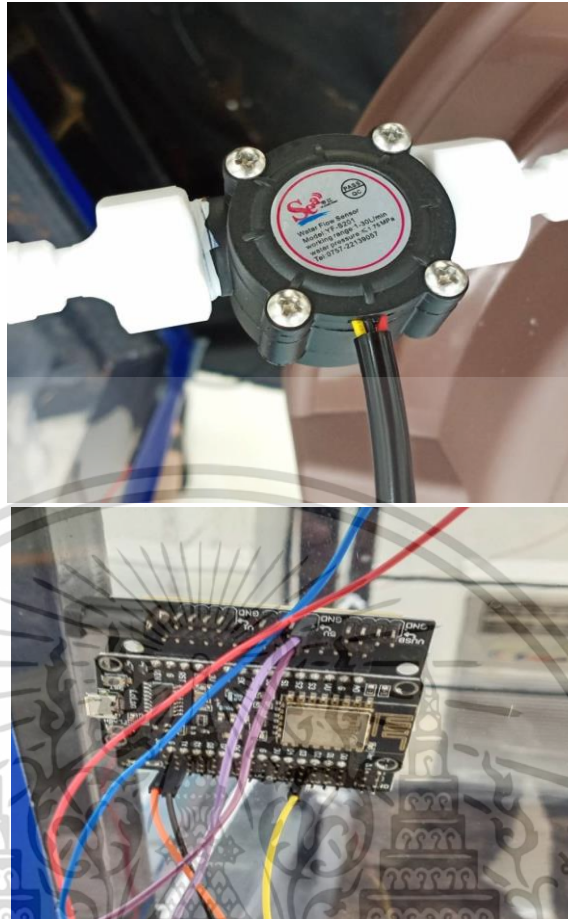


Figure 3.20 This is a figure of Water Flow Sensor

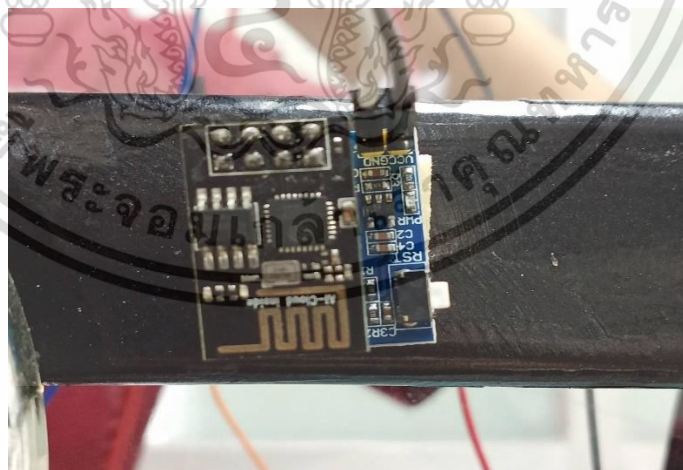


Figure 3.21 This is a figure of NodeMCU for ESP8266

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

3.7.3 Software

In the software part, XCode will take action to write the whole application as shown in figure 3.22. Swift is the computer language for XCode. Farmuto's application must connect with the Firebase database as shown in figure 3.23, so the data such as username will store on the cloud and can use it to identify the sensor's user.

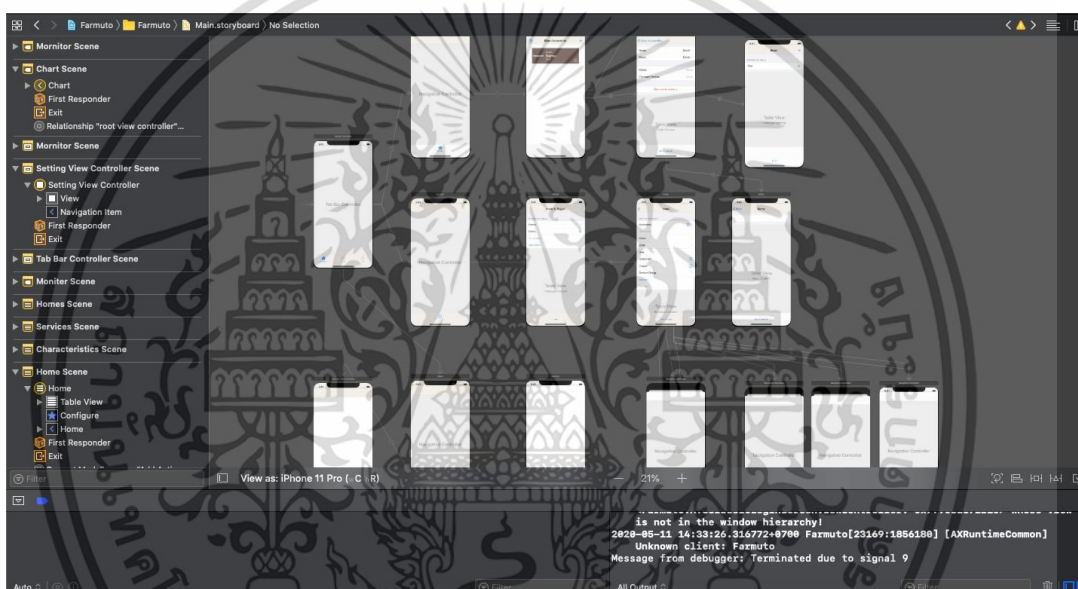


Figure 3.22 This is a figure of XCode

Arduino IDE or Arduino Integrated Development Environment is a software program that uses for written function in C and C++. This program is you for write program and upload any code from Arduino Board as shown in figure 3.24. This program is using for program Arduino board is to connect sensors and Arduino board. In addition, it is collected information from the sensors and send all the information back to the Firebase. Then, download the Arduino IDE driver into XCode and make all the data that the sensor collected show on the application.

Chapter 4

Experimental Results

This chapter review the result of this research following by the research methodology starting with the prototype's greenhouse, Tulip's plantation result, automatic system, hardware work, and application used result. The result of each following is:

4.1 Research Result

4.1.1 Greenhouse Result

The greenhouse can create its own environmental and control temperature. Humidity can be solved be a mini fan that set on the top of each side. In addition, there are no damage from insect and no weed growth in the greenhouse as shown in figure 4.1.

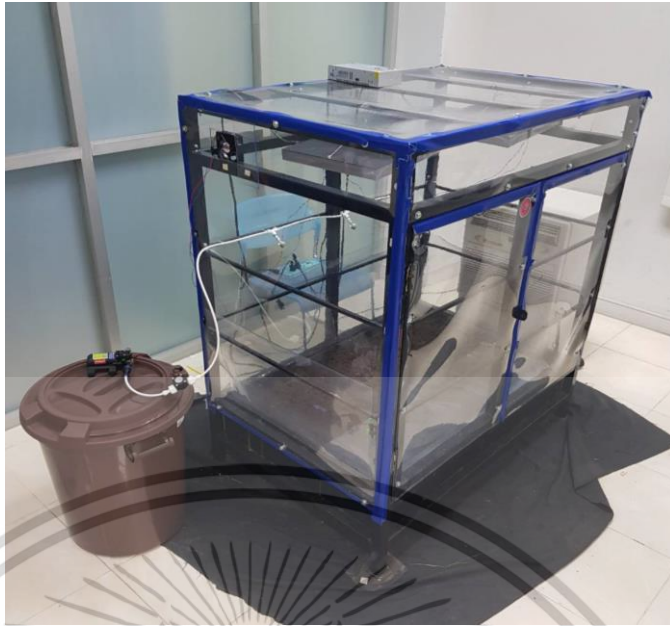


Figure 4.1 This is a figure of Greenhouse Box Complete Form

4.1.2 Light and Water Result

Light and water can be set a timer and command on/off via mobile application.

4.1.3 Application Result

Application is smooth and easy to use for anyone. Farmuto's application are work efficiency with no delay. All the user information that uses the application will be all store on the cloud firebase database as shown in figure 4.2.

4.1.4 Arduino Board Result

Arduino boards and sensors are working together and sending information into the application dashboard smoothly.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

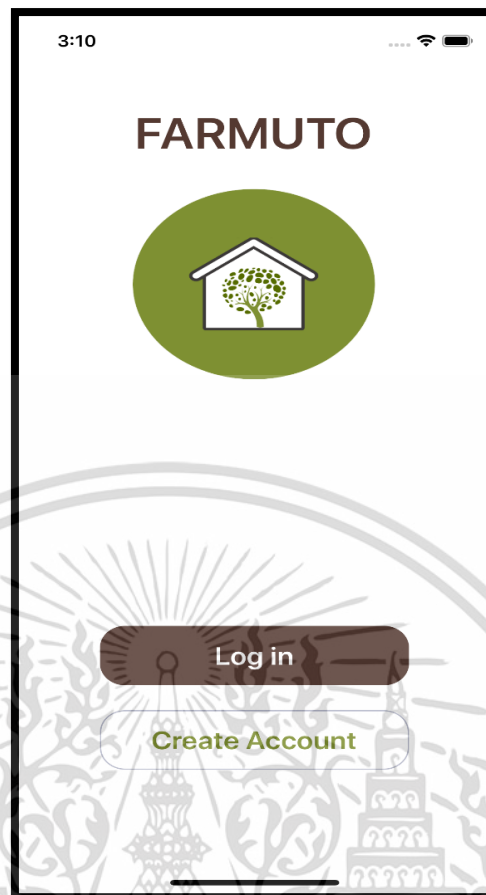


Figure 4.2 This is a figure of Complete Application

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

4.2 Use Case Diagram

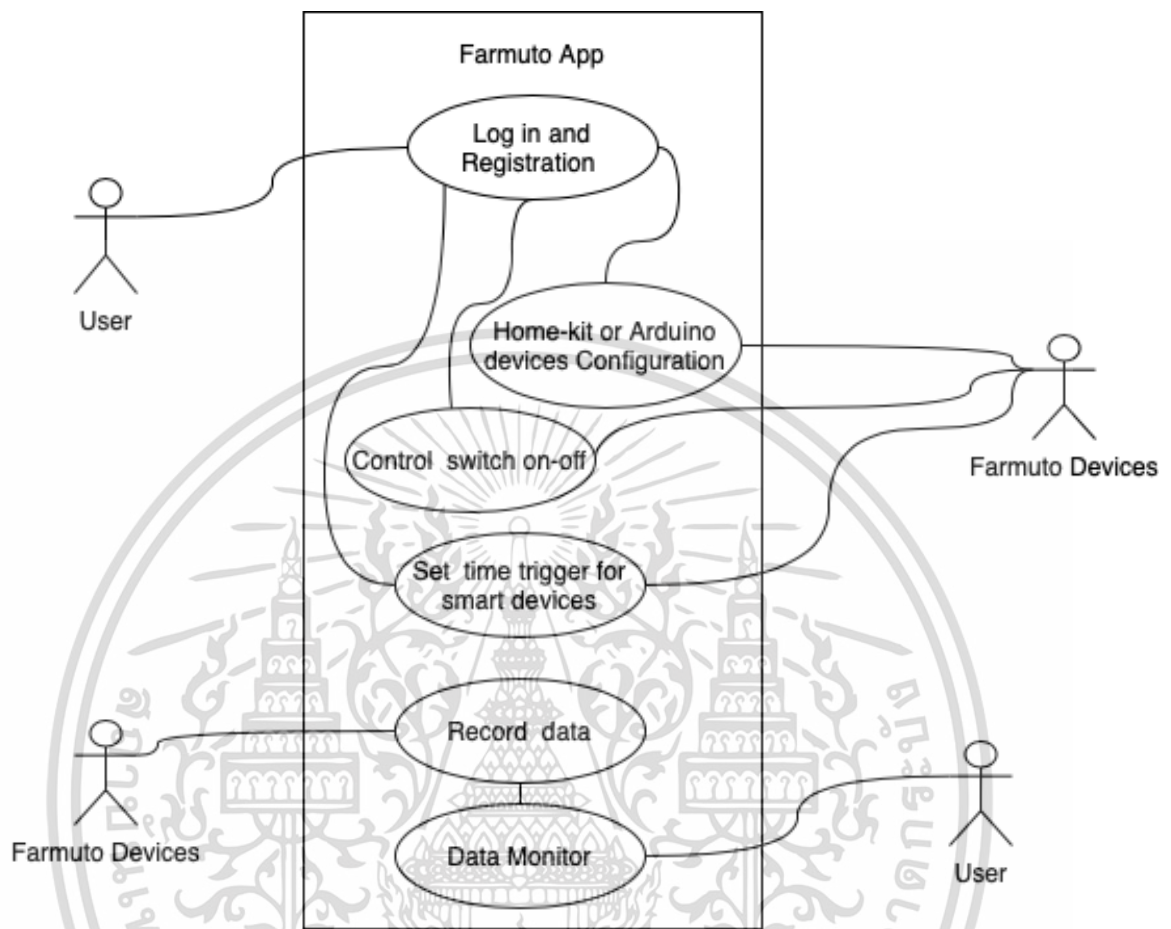


Figure 4.3 This is a figure of Use Case Diagram

4.2.1 Brief Use Case

The following are the description of the use cases in brief as shown in figure 4.3.

1) Log-in and Registration

Users can log-in and register their identification number.

2) Home-kit or Arduino Device Configuration

It is a plugin for XCode to connect Arduino IDE with XCode.

3) Control Switch On/Off

Users can turn on/off manually the light and water system via mobile application.

4) Set Time Trigger for Smart Device

User can set a timer for light and water system via mobile application.

5) Record data

The system will record user data from sensors into the firebase database.

6) Data Monitor

The system will show the collected data into the application graph board.

Chapter 5

Conclusions and Discussions

As a result, this project has achieved the objectives that the hypothesis had been set. The main objective of this project is to develop an IoT system using for agriculture. The system has been developed to suit an agriculture field by adding specific sensors that can be used for collecting agriculture data. The second objective is to develop an application used to control all devices in the IoT system. This objective can have been solving because the application can add more IoT devices from itself, instead of adding sensor by coding. The system works perfectly in every part starting from the application that can store data into firebase and command Arduino at the same time. The system does both job at the same time at a short time, which make the system have more consistency.

Sub-result of the project also succeeded in many ways. Firstly, the productivity and the quality are the best compared to the tulip that had plants outside the greenhouse. Tulip is growing faster inside the prototype greenhouse and most of them come in the same shape. On the other hand, they have a little difference in the leave and size due to the nutrition in the soil that might not perfectly equal and their genes. Secondly, the prototype greenhouse has been sealed in every corner which makes the environment inside the greenhouse completely different from outside. Due to the seal, it had almost

completely owned environmental in the greenhouse, only a few airs can flow through

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

the fan hole, but it has special purposes to decrease the humidity in the greenhouse. Lastly, the seal also protects and prevent insect or other intruder that can cause damage to the plant such as grasshopper, etc. Moreover, the greenhouse can prevent parasite plants and weed that can cause damage and steal nutrition from the tulip. The overall result, the Smart Farm system is quite successful, only a few uncontrol variables that make the result unsolved such as the size of the greenhouse and limited budget.

The main struggle of this project is the lack of tulip background information. Tulip is the flower that needs lots of care, so information like how many hours of light does it want or how much water does it take precisely is hard to find. It is difficult to make a controlled environment without any information.

This project is a concrete project that needs a lot of physical interaction to achieve the goal. Because of the Coronavirus situation that makes our group cannot go to university and make the project itself become one of the hardest struggles.

The suggestion of this project is to research more on background information about the plant. In order to grow a plant or increase the productivity of the plant is not rely on the system only, but the plant itself information is needed to create a system that serves that plant well enough.

Reference

- [1] Thailand, S. F. (n.d.). Smart Farm. [Online],
Available: <http://smartfarmthailand.com/precisionfarming/index.php/product/micro-climate-monitoring/83-smart-farm/>
- [2] Diakunwadee. (2018, December 17). UI vs. UX: อะไรคือความแตกต่างระหว่าง user interface และ user experience. [Online],
Available: <https://www.thaiprogrammer.org/2018/12/ui-vs-ux-อะไรคือความแตกต่างระหว่าง/>
- [3] ICT, D. (2019, May 29). UX / UI คืออะไร. [Online],
Available: <https://www.dmit.co.th/th/ข่าวสาร/ux-vs-ui/>.
- [4] A.B. (n.d.). NodeMCU Base Ver 1.0 for Nodemcu V3. [Online],
Available: <https://www.ab.in.th/product/103/nodemcu-base-ver-1-0-for-nodemcu-v3>.
- [5] Four, A. (2019, November 4). NodeMCU Base Ver 1.0 for ESP8266 NodeMCU V3 ESP8266 NodeMCU V3. [Online],
Available: <https://www.arduino4.com/product/737/nodemcu-base-ver-1-0-for-esp8266-nodemcu-v3-บอร์ดขยายขา-esp8266-nodemcu-v3>.
- [6] Tech, S. P. M. (2AD). Switching Power Supply 220V to 5V 600mA Hi-Link HLK-PM01. [Online],

Available: <https://www.spmicrotech.com/product/switching-power-supply-220v-to-5v-600ma-hlk-pm01/>.

[7] ARDUINO 1.8.12. (n.d.). [Online],

Available: <https://www.arduino.cc/en/Main/Software>.

[8] Vine, U. (n.d.). 11 Simple Guidelines To Know Before Buying Grow Lights.

[Online],

Available: <https://www.urbanvine.co/blog/11-core-guidelines-urban-farmers-must-know-before-buying-grow-lights>.

[9] Francis, R. (2019, September 12). Harvesting Smart Seeds: The Role of Data and AI in Agriculture. [Online],

Available: <https://iotnxt.com/ai-in-argiculture/>.

[10] พระจันทร์สัญจร. (2019, February 1). การปลูกดอกไฮยาซิน ทิวลิปในกรุง แบบง่ายๆ. [Online],

Available: <https://www.bloggang.com/mainblog.php?id=moving-moon>.

[11] Wanthai. (2019, February 1). การปลูกดอกไฮยาซิน ทิวลิปในกรุง แบบง่ายๆ. [Online],

Available: <https://www.bloggang.com/mainblog.php?id=moving-moon>.

[12] Types of Tulips. (2019, January 1). [Online],

Available: <http://strategycoaching.co/types-of-tulips>.

[13] Tulipa 'Foxtrot' (Double Early Tulip). (n.d.). [Online],

Available: <https://www.gardenia.net/plant/tulipa-foxtrot-double-early-tulip>.

[14] SmartFarm. (n.d.). Smart Farm DIY. [Online],

Available: <http://www.smartfarmdiy.com/>.

- [15] Co, P. (n.d.). ESP8266-01 Wi-Fi-RELAY MODULE. [Online],
Available: <https://www.partco.fi/en/electromechanics/relays/relay-modules/21116-esp-esp11002r.html>.
- [16] Stayhome, J. (n.d.). ความชื้นดินตรวจจับเครื่องวัดความชื้นเซ็นเซอร์โมดูลสำหรับการพัฒนา Arduino บอร์ด DIY ชิ้นส่วนสมาร์ต. [Online],
Available: <https://www.joom.com/th/products/5d1afc6f36b54d0101cd71e5>.
- [17] Elektronik, A. (n.d.). Arduino Compatible DHT11 Digital Temperature Humidity Sensor Mod. [Online],
Available: <https://www.asif-elektronik.ch/arduino-kompatibel-temperatur-und-feuchte-sensor-dht11-modul.html>.
- [18] 1/2 Flow Sensor For Volume Control Water Beer Milk Vending Machine Cheap Electronics Mobile Online Shopping. (n.d.). [Online],
Available: <https://www.dhgate.com/product/1-2-quot-flow-sensor-for-volume-control-water/493663428.html>.
- [19] หนึ่งฤทัย คูหาศักดิ์, & ศศ.ดร.ระพีพร ศรีจำปา. (n.d.). *Smart Farm Business Management for Small-to-Medium melon Business Enterprises (SMEs)*. มหาวิทยาลัยบูรพา.
- [20] ทรงเกียรติ อิงคามระชร, & อาณัติ หน่อทองคำ. (2018). *การพัฒนาเกษตรกรผู้ปลูกสับปะรดจังหวัดราชบุรีให้เป็น Smart Farmer โดยการเรียนรู้จาก Smart Farmer ต้นแบบ*. ชลบุรี: มหาวิทยาลัยราชภัฏหมู่บ้านจอมบึง.
- [21] สิตาวีร์ ชีววิรุฬห์. (2016). *สมาร์ตฟาร์ม (Smart Farm) การทำการเกษตรที่เป็นมิตรกับสิ่งแวดล้อม*. Bangkok: สำนักวิชาการ สำนักงานเลขาธิการสภาผู้แทนราษฎร.

- [22] บัณฑิตพงษ์ ศรีอานวย, สราวุธ แผลงสร, วีระสิทธิ์ ปิติเจริญพร, & พิมพ์ใจ สีหะนาม. (2019). *The Design of the Smart Farm System using the Internet of Things Technology for Lime Farms in Phetchaburi Province*. นครราชสีมา: วิทยาลัยนครราชสีมา.



เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และดัดแปลงอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Appendix

Farmuto Application Code



เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

All code of the application

FirebaseAuthManager

import FirebaseAuth

import UIKit

class FirebaseAuthManager {

func login(credential: AuthCredential, completionBlock: @escaping (_ success: Bool) -> Void) {

Auth.auth().signIn(with: credential, completion: { (firebaseUser, error) in

print(firebaseUser)

completionBlock(error == nil)

}}

}

func createUser(email: String, password: String, completionBlock: @escaping (_ success: Bool)

```

> Void) {

    Auth.auth().createUser(withEmail: email, password: password) {(authResult,
error) in

        if let user = authResult?.user {

            print(user)

            completionBlock(true)

        } else {

            completionBlock(true)

        }
    }

}

func signIn(email: String, pass: String, completionBlock: @escaping (_ success:
Bool) -> Void) {

    Auth.auth().signIn(withEmail: email, password: pass) { (result, error) in

        if let error = error, let _ = AuthErrorCode(rawValue: error._code) {

            completionBlock(false)

        } else {

```

```

        print(result)

        completionBlock(true)

    }

}

}

}

MornitorViveController
import UIKit
import Charts

class MornitorViewController: UIViewController, ChartViewDelegate{

    var shouldHideData: Bool = false

    @IBOutlet var chartView: LineChartView!

    @IBOutlet var chartView2: LineChartView!

    @IBOutlet var sliderX: UISlider!

```



```
@IBOutlet var sliderTextX: UITextField!
```

```
override func viewDidLoad() {
```

```
    navigationItem.title = "Chart"
```

```
    super.viewDidLoad()
```

```
    // Do any additional setup after loading the view.
```

```
    chartView.delegate = self
```

```
    chartView.chartDescription?.enabled = false
```

```
    chartView.dragEnabled = true
```

```
chartView.setScaleEnabled(true)
```

```
//chartView.pinchZoomEnabled = false
```

```
chartView.highlightPerDragEnabled = true
```

```
chartView.backgroundColor = UIColor(red: 134/255, green:  
202/255, blue: 227/255,  
alpha: 1)
```

```
chartView.legend.enabled = true
```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```
chartView.legend.font = .boldSystemFont(ofSize: 15)
```

```
let xAxis = chartView.xAxis
```

```
xAxis.labelPosition = .top
```

```
xAxis.labelFont = .systemFont(ofSize: 13, weight: .light)
```

```
xAxis.labelTextColor = .black
```

```
xAxis.drawAxisLineEnabled = false
```

```
xAxis.drawGridLinesEnabled = true
```

```
xAxis.centerAxisLabelsEnabled = true
```

```
xAxis.granularity = 3600
```

```
xAxis.valueFormatter = DateValueFormatter()
```

```
xAxis.labelFont = .boldSystemFont(ofSize: 13)
```

```
xAxis.axisLineColor = .black
```

```
xAxis.setLabelCount(4, force: true)
```

```
let leftAxis = chartView.leftAxis
```

```
leftAxis.labelPosition = .outsideChart
```

```
leftAxis.labelFont = .systemFont(ofSize: 13, weight: .light)
```

```
leftAxis.labelFont = .boldSystemFont(ofSize: 13)
```


`leftAxis.drawGridLinesEnabled = false`

`leftAxis.granularityEnabled = true`

`leftAxis.axisMinimum = 0`

`leftAxis.axisMaximum = 100`

`leftAxis.yOffset = -9`

`leftAxis.labelTextColor = .black`

`chartView.rightAxis.enabled = false`

```
chartView.legend.form = .line
```

```
sliderX.value = 100
```

```
slidersValueChanged(nil)
```

```
chartView.animate(xAxisDuration: 2.5)
```

```
chartView2.delegate = self
```

```
chartView2.chartDescription?.enabled = false
```

```
chartView2.dragEnabled = true
```

```
chartView2.setScaleEnabled(true)
```

```
//chartView2.pinchZoomEnabled = false
```

```
chartView2.highlightPerDragEnabled = true
```

```
chartView2.backgroundColor = UIColor(red: 93/255, green:  
64/255, blue: 55/255,
```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

alpha: 1)

```
chartView2.legend.enabled = true
```

```
chartView2.legend.font = .boldSystemFont(ofSize: 15)
```

```
let xAxis2 = chartView2.xAxis
```

```
xAxis2.labelPosition = .top
```

```
xAxis2.labelFont = .systemFont(ofSize: 13, weight: .light)
```

```
xAxis2.labelTextColor = .black
```

```
xAxis2.drawAxisLineEnabled = false
```

```
xAxis2.drawGridLinesEnabled = true
```

```
xAxis2.centerAxisLabelsEnabled = true
```

```
xAxis2.granularity = 3600
```

```
xAxis2.valueFormatter = DateValueFormatter()
```

```
xAxis2.labelFont = .boldSystemFont(ofSize: 13)
```

```
xAxis2.axisLineColor = .black
```

```
xAxis2.setLabelCount(4, force: true)
```



```
let leftAxis2 = chartView2.leftAxis
```

```
leftAxis2.labelPosition = .outsideChart
```

```
leftAxis2.labelFont = .systemFont(ofSize: 13, weight: .light)
```

```
leftAxis2.labelFont = .boldSystemFont(ofSize: 13)
```

```
leftAxis2.drawGridLinesEnabled = false
```

```
leftAxis2.granularityEnabled = true
```

```
leftAxis2.axisMinimum = 0
```

```
leftAxis2.axisMaximum = 80
```

```
leftAxis2.yOffset = -9
```

```
leftAxis2.labelTextColor = .black
```

```
chartView2.rightAxis.enabled = false
```

```
chartView2.legend.form = .line
```

```
sliderX.value = 100
```

```
slidersValueChanged(nil)
```

```
chartView2.animate(xAxisDuration: 2.5)
```

```
}
```

```
func updateChartData() {
```

```
    if self.shouldHideData {
```

```
        chartView.data = nil
```

```
        chartView2.data = nil
```

```
    }  
    return
```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```
self.setDataCount(Int(sliderX.value), range: 30)
```

```
}
```



เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```
func setDataCount(_ count: Int, range: UInt32) {
```

```
    let now = Date().timeIntervalSince1970
```

```
    let hourSeconds: TimeInterval = 3600
```

```
    let from = now - (Double(count) / 2) * hourSeconds
```

```
    let to = now + (Double(count) / 2) * hourSeconds
```

```
    let values1 = stride(from: from, to: to, by: hourSeconds).map { (x) ->  
        ChartDataEntry  
in
```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้


```
let y = Double.random(in: 16 ... 20)
```

```
return ChartDataEntry(x: x, y: Double(y))
```

```
}
```

```
let values2 = stride(from: from, to: to, by: hourSeconds).map { (x) ->  
ChartDataEntry
```

```
in
```

```
let y = Double.random(in: 80 ... 85)
```

```
return ChartDataEntry(x: x, y: Double(y))
```

```
}
```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```
let values3 = stride(from: from, to: to, by: hourSeconds).map { (x) ->
ChartDataEntry
```

```
in
```

```
let y = Double.random(in: 40 ... 60)
```

```
return ChartDataEntry(x: x, y: Double(y))
```

```
}
```

```
let set1 = LineChartDataSet(entries: values2, label: "Humidity")
```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```
set1.mode = .cubicBezier
```

```
set1.axisDependency = .left
```

```
set1.setColor(UIColor(red: 51/255, green: 181/255, blue: 229/255,  
alpha: 1))
```

```
set1.lineWidth = 3
```

```
set1.drawCirclesEnabled = false
```

```
set1.drawValuesEnabled = false
```

```
set1.fillAlpha = 0.26
```

```
set1.fillColor = UIColor(red: 51/255, green: 181/255, blue: 229/255,  
alpha: 1)
```

```
set1.highlightColor = UIColor(red: 244/255, green: 117/255, blue: 117/255, alpha: 1)
```

```
set1.drawCircleHoleEnabled = false
```

```
set1.drawHorizontalHighlightIndicatorEnabled = false
```

```
let set2 = LineChartDataSet(entries: values1, label: "Temperature")
```

```
set2.mode = .cubicBezier
```

```
set2.axisDependency = .left
```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```
set2.setColor(UIColor(red: 255/255, green: 218/255, blue: 185/255,  
alpha: 1))
```

```
set2.lineWidth = 3
```

```
set2.drawCirclesEnabled = false
```

```
set2.drawValuesEnabled = false
```

```
set2.fillAlpha = 0.26
```

```
set2.fillColor = UIColor(red: 51/255, green: 181/255, blue: 229/255,  
alpha: 1)
```

```
set2.highlightColor = UIColor(red: 244/255, green: 117/255, blue:  
117/255, alpha: 1)
```

```
set2.drawCircleHoleEnabled = false
```



```
let data = LineChartData(dataSets: [set1, set2])
```

```
data.setDrawValues(false)
```

```
data.setValueTextColor(.white)
```

```
data.setValueFont(.systemFont(ofSize: 9, weight: .light))
```

```
let set3 = LineChartDataSet(entries: values3, label: "Soil Humidity")
```

```
set3.axisDependency = .left
```

```
set3.setColor(UIColor(red: 255/255, green: 255/255, blue: 255/255,  
alpha: 1))
```

```
set3.lineWidth = 1.5
```

```
set3.drawCirclesEnabled = false
```

```
set3.drawValuesEnabled = false
```

```
set3.fillAlpha = 0.26
```

```
set3.fillColor = UIColor(red: 51/255, green: 181/255, blue: 229/255,  
alpha: 1)
```

```
set3.highlightColor = UIColor(red: 244/255, green: 117/255, blue:  
117/255, alpha: 1)
```

```
set3.drawCircleHoleEnabled = false
```

```
let data2 = LineChartData(dataSets: [set3])
```

```
data2.setValueTextColor(.white)
```

```
data2.setValueFont(.systemFont(ofSize: 9, weight: .light))
```

```
chartView.data = data
```

```
chartView2.data = data2
```

```

@IBAction func slidersValueChanged(_ sender: Any?) {

    sliderTextX.text = "\\(Int(sliderX.value))"

    self.updateChartData()

}

}

DateValueFormatter

import Foundation

import Charts

public class DateValueFormatter: NSObject, IAxisValueFormatter {

    private let dateFormatter = DateFormatter()

    override init() {

```

```

super.init()

dateFormatter.dateFormat = "dd MMM HH:mm"

}

public func stringForValue(_ value: Double, axis: AxisBase?) -> String {

    return dateFormatter.string(from: Date(timeIntervalSince1970: value))

}

}

SettingsSections

enum SettingsSection: Int, CaseIterable, CustomStringConvertible{

    case Graph

    case Account

    var description: String{

        switch self{

            case .Graph: return "Graph"

            case .Account: return "Account"

        }

```



```

    }

}

enum GraphOptions: Int, CaseIterable, CustomStringConvertible{

    case delete

    var description: String{

        switch self{

            case .delete: return "Delete Graph"

        }

    }

}

enum AccountOptions: Int, CaseIterable, CustomStringConvertible{

    case logout

    var description: String{

```

```

        case .logout: return "Log Out"

    }

}

}

```

SettingCell

```
import UIKit
```

```
class SettingsCell: UITableViewCell {
```

```
    // MARK: - Properties
```

```
    // MARK: - Init
```

```
    override init(style: UITableViewCell.CellStyle, reuseIdentifier: String?) {
```

```
        super.init(style: style, reuseIdentifier: reuseIdentifier)
```

```
    }
```

```
    required init?(coder aDecoder: NSCoder) {
```

```
fatalError("init(coder:) has not been implemented")

}

}
```

SettingViewController

```
import UIKit
```

```
private let reuseIdentifier = "SettingsCell"
```

```
class SettingViewController: UIViewController {
```

```
// MARK: - Properties
```

```
var tableView: UITableView!
```

```
var userInfoHeader: UserInfoHeader!
```

```
// MARK: - Init
```

```

override func viewDidLoad() {

    super.viewDidLoad()

    navigationItem.title = "Settings"

    if #available(iOS 13.0, *) {

        let navBarAppearance = UINavigationBarAppearance()

        navBarAppearance.configureWithOpaqueBackground()

        navBarAppearance.titleTextAttributes = [.foregroundColor: UIColor.black]

        navBarAppearance.largeTitleTextAttributes = [.foregroundColor:
UIColor.black]

        navBarAppearance.backgroundColor = UIColor.init(hexString: "F2EEE5")

        navigationController?.navigationBar.standardAppearance =
navBarAppearance

        navigationController?.navigationBar.scrollEdgeAppearance =
navBarAppearance

    }

    configureUI()

}

// MARK: - Helper Functions

```

```
func configureTableView() {  
  
    tableView = UITableView()  
  
    tableView.delegate = self  
  
    tableView.dataSource = self  
  
    tableView.rowHeight = 70  
  
    tableView.register(SettingsCell.self, forCellReuseIdentifier: reuseIdentifier)  
    view.addSubview(tableView)  
    tableView.frame = view.frame  
  
    let frame = CGRect(x: 0, y: 88, width: view.frame.width, height: 100)  
    userInfoHeader = UserInfoHeader(frame: frame)  
  
    tableView.tableHeaderView = userInfoHeader  
  
    tableView.tableFooterView = UIView()  
  
}
```

```
func configureUI() {
```



```

configureTableView()

navigationController?.navigationBar.isTranslucent = false

navigationController?.navigationBar.barStyle = .black

navigationController?.navigationBar.barTintColor = UIColor(hexString:
"543931")

}

}

extension SettingViewController: UITableViewDelegate, UITableViewDataSource
{

func numberOfSections(in tableView: UITableView) -> Int {

return SettingsSection.allCases.count

}

func tableView(_ tableView: UITableView, numberOfRowsInSection section:
Int) -> Int {

```

```

guard let section = SettingsSection(rawValue: section) else { return 0 }

switch section {

case .Graph: return GraphOptions.allCases.count

case .Account: return AccountOptions.allCases.count

}
}

func tableView(_ tableView: UITableView, viewForHeaderInSection section: Int)
-> UIView? {

let view = UIView()

view.backgroundColor = UIColor(hexString: "543931")

print("Section is \(section)")

let title = UILabel()

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

title.font = UIFont.boldSystemFont(ofSize: 16)

title.textColor = .white

title.text = SettingsSection(rawValue: section)?.description

view.addSubview(title)

title.translatesAutoresizingMaskIntoConstraints = false

title.centerYAnchor.constraint(equalTo: view.centerYAnchor).isActive = true

title.leftAnchor.constraint(equalTo: view.leftAnchor, constant: 16).isActive =
true

return view
}

func tableView(_ tableView: UITableView, heightForHeaderInSection section:
Int) -> CGFloat {

return 40

}

func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath)
->
UITableViewCell {

```

```
let cell = tableView.dequeueReusableCell(withIdentifier: reuseIdentifier, for:
indexPath) as!
```

```
SettingsCell
```

```
guard let section = SettingsSection(rawValue: indexPath.section) else { return
UITableViewCell() }
```

```
switch section {
```

```
case .Graph:
```

```
let communications = GraphOptions(rawValue: indexPath.row)
```

```
cell.textLabel?.text = communications?.description
```

```
case .Account:
```

```
let social = AccountOptions(rawValue: indexPath.row)
```

```
cell.textLabel?.text = social?.description
```

```
}
```

```
return cell
```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

    }

    func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {

        guard let section = SettingsSection(rawValue: indexPath.section) else { return
        }

        switch section {

        case .Graph:

            print(GraphOptions(rawValue: indexPath.row)?.description)

        case .Account:

            print(AccountOptions(rawValue: indexPath.row)?.description)

            let view = ATCClassicLandingScreenViewController(nibName: nil, bundle:
            nil)

            self.hidesBottomBarWhenPushed = true

            self.navigationController?.pushViewController(view,animated: true)

        }

    }

}

}

```


UserInfoHeader

import UIKit

class UserInfoHeader: UIView {

// MARK: - Properties

let profileImageView: UIImageView = {

let iv = UIImageView()

iv.contentMode = .scaleAspectFill

iv.clipsToBounds = true

iv.translatesAutoresizingMaskIntoConstraints = false

iv.image = UIImage(named: "guide")

return iv

}()

let usernameLabel: UILabel = {

let label = UILabel()

```

        label.text = "Guidebth"

        label.font = UIFont.systemFont(ofSize: 18)

        label.translatesAutoresizingMaskIntoConstraints = false

        return label

    }

    let emailLabel: UILabel = {

        let label = UILabel()

        label.text = "guidebth@gmail.com"

        label.font = UIFont.systemFont(ofSize: 16)

        label.textColor = .lightGray

        label.translatesAutoresizingMaskIntoConstraints = false

        return label

    }

    // MARK: - Init

```

```

    override init(frame: CGRect) {

```

```

super.init(frame: frame)

let profileImageDimension: CGFloat = 60

addSubview(profileImageView)

profileImageView.centerYAnchor.constraint(equalTo:
centerYAnchor).isActive = true

profileImageView.leftAnchor.constraint(equalTo: leftAnchor, constant:
16).isActive = true

profileImageView.widthAnchor.constraint(equalToConstant:
profileImageDimension).isActive
= true

profileImageView.heightAnchor.constraint(equalToConstant:
profileImageDimension).isActive = true

profileImageView.layer.cornerRadius = profileImageDimension / 2

addSubview(usernameLabel)

usernameLabel.centerYAnchor.constraint(equalTo:
profileImageView.centerYAnchor,

```

```
constant: -10).isActive = true
```

```
usernameLabel.leftAnchor.constraint(equalTo:  
profileImageView.rightAnchor, constant:
```

```
12).isActive = true
```

```
addSubview(emailLabel)
```

```
emailLabel.centerYAnchor.constraint(equalTo:  
profileImageView.centerYAnchor, constant:
```

```
10).isActive = true
```

```
emailLabel.leftAnchor.constraint(equalTo: profileImageView.rightAnchor,  
constant:
```

```
12).isActive = true
```

```
}
```

```
required init?(coder aDecoder: NSCoder) {
```

```
fatalError("init(coder:) has not been implemented")
```

```
}
```

```
}
```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

AccessorySetting

import UIKit

import HomeKit

class AccessorySettings: UITableViewController {

/// The list of characteristics to display.

var characteristics = [HMCharacteristic]()

/// The service that owns the characteristics to display.

var service: HMService? {

didSet {

// Capture a curated list of characteristics.

characteristics = service?.displayableCharacteristics ?? []

}

}


```
/// Registers this instance to receive accessory delegate callbacks.

override func viewDidLoad() {

    super.viewDidLoad()

    HomeStore.shared.addAccessoryDelegate(self)

}

/// Deregisters this instance as an accessory delegate.
deinit {
    HomeStore.shared.removeAccessoryDelegate(self)
}

// MARK: - Table view data source

/// Tells the table view to create one cell for each characteristic.

override func tableView(_ tableView: UITableView, numberOfRowsInSectionSection
section: Int) -> Int {
```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

        return characteristics.count
    }

    /// Supplies the table view with configured cells.

    override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) ->
UITableViewCell {

        let cell = tableView.dequeueReusableCell(withIdentifier: "CharacterCell", for:
indexPath)

        if let characterCell = cell as? CharacterCell {
            characterCell.characteristic = characteristics[indexPath.row]
        }

        return cell
    }
}

CharaterCell

```

```
import UIKit

import HomeKit

class CharacterCell: UITableViewCell {

    @IBOutlet weak var slider: UISlider!

    @IBOutlet weak var toggle: UISwitch!

    @IBOutlet weak var nameLabel: UILabel!

    @IBOutlet weak var valueLabel: UILabel!

    @IBOutlet weak var segmentedControl: UISegmentedControl!

    /// Writes to the characteristic as the user moves the slider back and forth.

    @IBAction func sliderDidChange(_ sender: UISlider) {

        guard let characteristic = characteristic else { return }

        // Convert to a value in range, and on the grid.

        let value = characteristic.valueFor(fraction: sender.value)
```

```

// Only write when there's a change.

guard

    let oldValue = characteristic.decimalValue,

    value != oldValue else { return }

// Write the new value.

characteristic.writeValue(value) { error in

    if let error = error {

        print(error.localizedDescription)

    } else {

        // Update the label to match.

        self.redrawValueLabel()

    }

}

}

}

/// Ensures the final state of the UI matches reality after the user finishes sliding.

@IBAction func sliderTouchUp(_ sender: Any) {

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า

ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

readAndRedrawValue(animated: false)

}

/// Updates the characteristic value when the toggle switch changes state.

@IBAction func toggleDidChange(_ sender: UISwitch) {

    guard let characteristic = characteristic else { return }

    characteristic.writeValue(sender.isOn) { error in
        if let error = error {
            print(error.localizedDescription)
        }
    }
}

}

/// Updates the characteristic value when the segmented control changes state.

@IBAction func segDidChange(_ sender: UISegmentedControl) {

    guard let characteristic = characteristic else { return }

```



```

characteristic.writeValue(sender.selectedSegmentIndex) { error in

    if let error = error {

        print(error.localizedDescription)

    }

}

}

}

// The characteristic that this cell displays.
var characteristic: HMCharacteristic? {

    // Update the UI when a new characteristic is set.
    didSet {

        // A cell with no characteristic shows as empty.

        guard

            let characteristic = characteristic,

            let accessory = characteristic.service?.accessory else {

                nameLabel.text = nil

                valueLabel.text = nil

```

```

        slider.isHidden = true

        toggle.isHidden = true

        return

    }

    // Set the characteristic's name
    nameLabel.text = characteristic.displayName

    // Show the toggle for reachable, writeable, Boolean characteristics.
    toggle.isHidden = !accessory.isReachable || !characteristic.isBool ||
!characteristic.isWriteable
    toggle.isOn = false

    // Show the segmented control for writeable values with declared state names.

    segmentedControl.isHidden = !(accessory.isReachable &&
characteristic.isWriteable &&
!characteristic.stateNames.isEmpty)

    if !segmentedControl.isHidden {

```

```

segmentedControl.removeAllSegments()

for index in 0...(characteristic.stateNames.count - 1) {

    let name = characteristic.stateNames[index]

    segmentedControl.insertSegment(withTitle: name, at: index, animated:
false)

}

}

// Show the slider for other numbers.

// slider.isHidden = !(segmentedControl.isHidden && accessory.isReachable
&&
characteristic.isNumeric && characteristic.stateNames.isEmpty)

// slider.isEnabled = characteristic.isWriteable

// slider.value = 0


// Value label shows when toggle and segment are hidden.

valueLabel.isHidden = !(toggle.isHidden && segmentedControl.isHidden)

valueLabel.text = "—"

```

```

// Execute a read, and then update the UI with it.

readAndRedrawValue(animated: false)

}

}

/// Reads the characteristic value from the HomeKit database, and updates the UI.
func readAndRedrawValue(animated: Bool) {
    guard
        let characteristic = characteristic,
        characteristic.isReadable,
        let accessory = characteristic.service?.accessory,
        accessory.isReachable else { return }

    characteristic.readValue { error in

        if let error = error {

            print(error.localizedDescription)

        } else {

```

```

        self.redrawValueLabel()

        self.redrawControls(animated: animated)

    }

}

}

/// Sets the cell's secondary text according to the current characteristic value.
func redrawValueLabel() {
    valueLabel.text = characteristic?.formattedValueString
}

/// Sets any visible graphical control to the current characteristic value.
func redrawControls(animated: Bool) {

    guard

        let characteristic = characteristic,

        let value = characteristic.value else { return }

    if !slider.isHidden {

```



```

if let decimalValue = characteristic.decimalValue {

    let fraction = (decimalValue / characteristic.span) as NSNumber

    slider.setValue(fraction.floatValue, animated: animated)

}

} else if !toggle.isHidden {

    if let boolValue = value as? Bool {

        toggle.setOn(boolValue, animated: animated)

    }

} else if !segmentedControl.isHidden {

    if let intValue = value as? Int {

        segmentedControl.selectedSegmentIndex = intValue

    }

}

}

}

}

}

```

AccessorySetting+AccessoryDelegate

```

import HomeKit

extension AccessorySettings: HMAccessoryDelegate {

    /// Handles characteristic updates.

    func accessory(_ accessory: HMAccessory, service: HMService,
didUpdateValueFor
characteristic: HMCharacteristic) {

        guard service == self.service else { return }

        // Find the cell that displays this characteristic.
        guard let row = characteristics.firstIndex(of: characteristic),

            let cell = tableView.cellForRow(at: IndexPath(row: row, section: 0)) as?
CharacterCell

            else { return }

        // Tell the cell to refresh using the characteristic it already knows about.

        cell.redrawValueLabel()

        cell.redrawControls(animated: true)

```

```

    }

}

AccessoryDetail+HomeDelegate

import HomeKit

/// Handle the home delegate callbacks.
extension AccessoryDetail: HMHomeDelegate {

    func home(_ home: HMHome, didUpdate room: HMRoom, for accessory:
HMAccessory) {

        reloadData()

    }

    func home(_ home: HMHome, didUpdateNameFor room: HMRoom) {

        reloadData()

    }

}

}

AccessoryDetail

```

```
import UIKit

import HomeKit

class AccessoryDetail: UITableViewController {

    @IBOutlet weak var nameLabel: UILabel!

    @IBOutlet weak var roomLabel: UILabel!

    @IBOutlet weak var modelLabel: UILabel!

    @IBOutlet weak var firmwareLabel: UILabel!

    /// The service that this detail view shows. The UI refers to HMService
    /// instances as 'accessories' to be consistent with Home app terminology.

    var service: HMService?

    /// The home of which the service is a part, used for room picking and accessory
    removal.

    var home: HMHome?
```

```

/// Resets the view data based on the current service.

func reloadData() {

    title = service?.name ?? "Accessory Detail"

    nameLabel.text = service?.name

    roomLabel.text = service?.accessory?.room?.name

    modelLabel.text = service?.accessory?.model

    firmwareLabel.text = service?.accessory?.firmwareVersion

}

/// Registers this view controller to receive various delegate callbacks.
override func viewDidLoad() {

    super.viewDidLoad()

    HomeStore.shared.addHomeDelegate(self)

    HomeStore.shared.addAccessoryDelegate(self)

    reloadData()

}

```



```
/// Deregisters this view controller as various kinds of delegate.
```

```
deinit {
```

```
    HomeStore.shared.removeHomeDelegate(self)
```

```
    HomeStore.shared.removeAccessoryDelegate(self)
```

```
}
```

```
/// Prepares to show one of the child views of this view.
```

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
```

```
    if let controller = segue.destination as? NameEditor {
```

```
        controller.service = service
```

```
    } else if let controller = segue.destination as? RoomPicker {
```

```
        controller.service = service
```

```
        controller.home = home
```

```
    } else if let controller = segue.destination as? AccessorySettings {
```

```
        controller.service = service
```

```

    }

}

/// Handles table view cell taps.

override func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {

    if indexPath == IndexPath(row: 0, section: 2) {

        let alert = UIAlertController(title: "Remove Accessory",
                                     message: "Are you sure you want to remove this
accessory?",
                                     preferredStyle: .alert)

        alert.addAction(UIAlertAction(title: "Remove", style: .destructive) { _ in
            if let accessory = self.service?.accessory,

                let home = self.home {

                HomeStore.shared.remove(accessory, from: home)

            }

        })

        alert.addAction(UIAlertAction(title: "Cancel", style: .cancel) { _ in

```

```

        self.tableView.deselectRow(at: indexPath, animated: true)

    })

    present(alert, animated: true)

}

}

}

AccessoryDetail+AccessoryDelegate
import HomeKit

/// Handle the accessory delegate callbacks.
extension AccessoryDetail: HMAccessoryDelegate {

    func accessory(_ accessory: HMAccessory, didUpdateNameFor service:
HMService) {

        reloadData()

    }

}

}

ControlsViewController

import UIKit

```

```

import UIKit

/// A view controller which displays a list of HMServices, separated by Service Type.

class ControlsViewController: HMCatalogViewController, HMAccessoryDelegate
{

    // MARK: Types

    struct Identifiers {

        static let showServiceSegue = "Show Service"

    }

    // MARK: Properties

    var tableViewDataSource: ControlsTableViewDataSource!

    var cellController = AccessoryUpdateController()

    @IBOutlet weak var addButton: UIBarButtonItem!

```

```

// MARK: View Methods

/// Sends the selected service into the destination view controller.

override func prepare(for segue: UIStoryboardSegue, sender: Any?) {

    super.prepare(for: segue, sender: sender)

    if segue.identifier == Identifiers.showServiceSegue {

        if let indexPath = tableView.indexPath(for: sender as! UITableViewCell) {

            let characteristicsViewController = segue.intendedDestinationViewController as! CharacteristicsViewController

            if let selectedService = tableViewDataSource.serviceForIndexPath(indexPath) {

                characteristicsViewController.service = selectedService

            }

            characteristicsViewController.cellDelegate = cellController

        }
    }
}

```



```

    }

}

/// Initializes the table view data source.

override func viewDidLoad() {

    super.viewDidLoad()

    tableViewDataSource = ControlsTableViewDataSource(tableView: tableView)

}

/// Reloads the view.

override func viewWillAppear(_ animated: Bool) {

    super.viewWillAppear(animated)

    navigationItem.title = home.name

    reloadData()

}

// MARK: Helper Methods

```

```

private func reloadData() {

    tableViewDataSource.reloadData()

    let sections = tableViewDataSource.numberOfSections(in: tableView)

    if sections == 0 {

        setBackgroundMessage(tableViewDataSource.emptyMessage())

    }
    else {

        setBackgroundMessage(nil)

    }
}

// MARK: Delegate Registration

/// Registers as the delegate for the current home and all accessories in the home.

override func registerAsDelegate() {

    super.registerAsDelegate()

    for accessory in home.accessories {

```

```

        accessory.delegate = self

    }

}

/*

Any delegate methods which could change data will reload the
table view data source.

*/

// MARK: HMHomeDelegate Methods

func home(_ home: HMHome, didAddAccessory accessory: HMAccessory) {

    accessory.delegate = self

    reloadData()

}

func home(_ home: HMHome, didRemoveAccessory accessory: HMAccessory)
{

```

```

        reloadData()

    }

// MARK: HMAccessoryDelegate Methods

func accessoryDidUpdateReachability(_ accessory: HMAccessory) {

    reloadData()

}

func accessory(_ accessory: HMAccessory, didUpdateNameFor service:
HMService) {

    reloadData()

}

func accessory(_ accessory: HMAccessory, didUpdateAssociatedServiceTypeFor
service:

HMService) {

    reloadData()

```

```

    }

    func accessoryDidUpdateServices(_ accessory: HMAccessory) {

        reloadData()

    }

    func accessoryDidUpdateName(_ accessory: HMAccessory) {

        reloadData()

    }

}

ControlsTableViewDataSource

import UIKit

import HomeKit

/// A UITableViewDataSource that populates the table in ControlsViewController.

class ControlsTableViewDataSource: NSObject, UITableViewDataSource {

    // MARK: Types

```



```
struct Identifiers {  
  
    static let serviceCell = "ServiceCell"  
  
    static let unreachableServiceCell = "UnreachableServiceCell"  
  
}
```

```
// MARK: Properties
```

```
var serviceTable: [String: [HMService]]?
```

```
var sortedKeys: [String]?
```

```
let tableView: UITableView
```

```
var home: HMHome? {
```

```
    return HomeStore.shared.home
```

```
}
```

```
/// Initializes the table view and data source.
```

```
required init(tableView: UITableView) {
```

```
    self.tableView = tableView
```

```

super.init()

self.tableView.dataSource = self

}

/**
Reloads the table, sets the table's dataSource to self,
regenerated the service table, creates a sorted list of keys,
sets the home's delegate, and reloads the table.
*/
func reloadData() {
    if let home = home {
        serviceTable = home.serviceTable

        sortedKeys = serviceTable!.keys.sorted()

        tableView.reloadData()

    }

    else {

        serviceTable = nil

        sortedKeys = nil

```

```

    }

    tableView.reloadData()

}

/// - returns: The localized description of the service type for that section.

func tableView(_ tableView: UITableView, titleForHeaderInSection section: Int)
-> String? {

    return sortedKeys?[section]

}

func numberOfSections(in tableView: UITableView) -> Int {

    return sortedKeys?.count ?? 0

}

/**

- returns: A message that corresponds to the current most important reason

that there are no services in the table. Either "No Accessories"

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

        or "No Services".

    */

    func emptyMessage() -> String {

        if home?.accessories.count == 0 {

            return NSLocalizedString("No Accessories", comment: "No Accessories")

        }

        else {

            return NSLocalizedString("No Services", comment: "No Services")

        }

    }

    /// - returns: The number of services matching the service type in that section.

    func tableView(_ tableView: UITableView, numberOfRowsInSection section:
    Int) -> Int {

        return serviceTable![sortedKeys![section]]!.count

    }

    /// - returns: A ServiceCell set for the service at the provided index path.

```

```

func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath)
->

UITableViewCell {

    let service = serviceForIndexPath(indexPath)!

    let reuseIdentifier = service.accessory!.isReachable ? Identifiers.serviceCell :
Identifiers.unreachableServiceCell

    let cell = tableView.dequeueReusableCell(withIdentifier: reuseIdentifier, for:
indexPath) as!
ServiceCell

    cell.service = service

    return cell

}

```

/// - returns: The service represented at the index path in the table.


```

func serviceForIndexPath(_ indexPath: IndexPath) -> HMService? {

    if let sortedKeys = sortedKeys,

        let serviceTable = serviceTable,

        let services = serviceTable[sortedKeys[indexPath.section]] {

        return services[indexPath.row]

    }

    return nil
}

}

AccessoryBrowserViewContrller

import UIKit

import HomeKit

/// A view controller that displays a list of characteristics within an HMService.

class CharacteristicsViewController: HMCatalogViewController,
HMAccessoryDelegate {

```

```
// MARK: Properties
```

```
var service: HMService!
```

```
var cellDelegate: CharacteristicCellDelegate!
```

```
private var tableViewDataSource: CharacteristicsTableViewDataSource!
```

```
var showsFavorites = false
```

```
var allowsAllWrites = false
```

```
// MARK: View Methods
```

```
/// Initializes the data source.
```

```
override func viewDidLoad() {
```

```
    super.viewDidLoad()
```

```
    tableViewDataSource = CharacteristicsTableViewDataSource(service: service,  
    tableView:
```

```
    tableView,    delegate:    cellDelegate,    showsFavorites:    showsFavorites,  
    allowsAllWrites:
```

```
allowsAllWrites)
```

```
}
```

```
/// Reloads the view and enabled notifications for all relevant characteristics.
```

```
override func viewWillAppear(_ animated: Bool) {
```

```
    super.viewWillAppear(animated)
```

```
    title = service.name
```

```
    setNotificationsEnabled(true)
```

```
    reloadDataTableView()
```

```
}
```

```
/// Disables notifications for characteristics.
```

```
override func viewWillDisappear(_ animated: Bool) {
```

```
    super.viewWillDisappear(animated)
```

```
    setNotificationsEnabled(false)
```

```
}
```

```
/**
```

Registers as the delegate for the current home and

the service's accessory.

*/

override func registerAsDelegate() {

super.registerAsDelegate()

service.accessory?.delegate = self

}

/**

Enables or disables notifications on all characteristics within this service.

- parameter notificationsEnabled: A Bool; whether to enable or disable.

*/

func setNotificationsEnabled(_ notificationsEnabled: Bool) {

for characteristic in service.characteristics {

if characteristic.supportsEventNotification {

characteristic.enableNotification(notificationsEnabled) { error in

if let error = error {


```

override func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {

    switch CharacteristicTableViewSection(rawValue: indexPath.section) {

        case .characteristics?:

            let characteristic = service.characteristics[indexPath.row]

            didSelectCharacteristic(characteristic, atIndexPath: indexPath)

            case .associatedServiceType?:

                didSelectAssociatedServiceTypeAtIndexPath(indexPath)

            case nil:

                fatalError("Unexpected CharacteristicTableViewSection raw value.")

        }

    }

}

/**

```

If a characteristic is selected, and it is the 'Identify' characteristic,

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

perform an identify on that accessory.

*/

```
private func didSelectCharacteristic(_ characteristic: HMCharacteristic,
atIndexPath indexPath:
```

```
IndexPath) {
```

```
    if characteristic.isIdentify {
```

```
        service.accessory?.identify { error in
```

```
            if let error = error {
```

```
                self.displayError(error)
```

```
            }
            return
```

```
        }
```

```
    }
```

```
}
```

```
}
```

```
/**
```

Handles selection of one of the associated service types in the list.

- parameter indexPath: The selected index path.

*/

```
private func didSelectAssociatedServiceTypeAtIndexPath(_ indexPath:
IndexPath) {
```

```
    let serviceTypes = HMService.validAssociatedServiceTypes
```

```
    var newServiceType: String?
```

```
    if indexPath.row < serviceTypes.count {
```

```
        newServiceType = serviceTypes[indexPath.row]
```

```
    }
```

```
    service.updateAssociatedServiceType(newServiceType) { error in
```

```
        if let error = error {
```

```
            self.displayError(error)
```

```
            return
```

```
        }
```

```
        self.didUpdateAssociatedServiceType()
```

```
    }
```

```
}
```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

/// Reloads the associated service section in the table view.

private func didUpdateAssociatedServiceType() {

    let associatedServiceTypeIndexSet = IndexSet(integer:
CharacteristicTableViewSection.associatedServiceType.rawValue)

    tableView.reloadSections(associatedServiceTypeIndexSet, with: .automatic)
}

// MARK: HMHomeDelegate Methods

/// If our accessory was removed, pop to root view controller.

func home(_ home: HMHome, didRemoveAccessory accessory: HMAccessory) {

    if accessory == service.accessory {

        _ = navigationController?.popToRootViewController(animated: true)

    }

}

```

```

// MARK: HMAccessoryDelegate Methods

/// If our accessory becomes unreachable, pop to root view controller.

func accessoryDidUpdateReachability(_ accessory: HMAccessory) {

    if accessory == service.accessory && !accessory.isReachable {

        _ = navigationController?.popToRootViewController(animated: true)

    }

}

/**
Search for the cell corresponding to that characteristic and
update its value.

*/

func accessory(_ accessory: HMAccessory, service: HMService,
didUpdateValueFor
characteristic: HMCharacteristic) {

    if let index = service.characteristics.firstIndex(of: characteristic) {

        let indexPath = IndexPath(row: index, section: 0)

```



```

        if let cell = tableView.cellForRow(at: indexPath) as? CharacteristicCell {

            cell.setValue(characteristic.value as? CellValueType, notify: false)

        }

    }

}

```

CharacteristicsViewController

```
import UIKit
```

```
import HomeKit
```

```
/// Represents the sections in the CharacteristicsViewController.
```

```
enum CharacteristicTableViewSection: Int {
```

```
    case characteristics, associatedServiceType
```

```
}
```

```
/// A UITableViewDataSource that populates a CharacteristicsViewController.
```

```
class CharacteristicsTableViewDataSource: NSObject, UITableViewDelegate,
```

```
UITableViewDataSource {
```

```

// MARK: Types

struct Identifiers {

    static let characteristicCell = "CharacteristicCell"

    static let sliderCharacteristicCell = "SliderCharacteristicCell"

    static let switchCharacteristicCell = "SwitchCharacteristicCell"

    static let segmentedControlCharacteristicCell =
"SegmentedControlCharacteristicCell"

    static let textCharacteristicCell = "TextCharacteristicCell"

    static let serviceTypeCell = "ServiceTypeCell"

}

// MARK: Properties

var service: HMService

var tableView: UITableView

var delegate: CharacteristicCellDelegate

var showsFavorites: Bool

```

```
var allowsAllWrites: Bool
```

```
/// Sets up properties from specified values, configures the table view, and cell  
reuse identifiers.
```

```
required init(service: HMService, tableView: UITableView, delegate:  
CharacteristicCellDelegate,
```

```
showsFavorites: Bool = false, allowsAllWrites: Bool = false) {
```

```
    self.service = service
```

```
    self.tableView = tableView
```

```
    self.delegate = delegate
```

```
    self.showsFavorites = showsFavorites
```

```
    self.allowsAllWrites = allowsAllWrites
```

```
    super.init()
```

```
    tableView.dataSource = self
```

```
    tableView.rowHeight = UITableView.automaticDimension
```

```
    tableView.estimatedRowHeight = 50.0
```

```
    registerReuseIdentifiers()
```

```
}
```

```

/// Registers all of the characteristic cell reuse identifiers with this table.

func registerReuseIdentifiers() {

    let characteristicNib = UINib(nibName: Identifiers.characteristicCell, bundle:
nil)

    tableView.register(characteristicNib,                                forCellReuseIdentifier:
Identifiers.characteristicCell)

    let sliderNib = UINib(nibName: Identifiers.sliderCharacteristicCell, bundle:
nil)

    tableView.register(sliderNib,                                forCellReuseIdentifier:
Identifiers.sliderCharacteristicCell)

    let switchNib = UINib(nibName: Identifiers.switchCharacteristicCell, bundle:
nil)

    tableView.register(switchNib,                                forCellReuseIdentifier:
Identifiers.switchCharacteristicCell)

    let segmentedNib = UINib(nibName: Identifiers.segmentedControlCharacteristicCell, bundle:
nil)

```

```

tableView.register(segmentedNib, forCellReuseIdentifier:
Identifiers.segmentedControlCharacteristicCell)

let textNib = UINib(nibName: Identifiers.textCharacteristicCell, bundle: nil)

tableView.register(textNib, forCellReuseIdentifier:
Identifiers.textCharacteristicCell)

tableView.register(UITableViewCell.self, forCellReuseIdentifier:
Identifiers.serviceTypeCell)
}

/**
- returns: The number of sections, computed from whether or not
the services supports an associated service type.

*/

func numberOfSections(in tableView: UITableView) -> Int {

return service.supportsAssociatedServiceType ? 2 : 1

}

```


/**

The characteristics section uses the services count to generate the number of rows.

The associated service type uses the valid associated service types.

*/

```
func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
```

```
    switch CharacteristicTableViewSection(rawValue: section) {
```

```
        case .characteristics?:
```

```
            return service.characteristics.count
```

```
        case .associatedService?:
```

```
            // For 'None'.
```

```
            return HMService.validAssociatedServiceTypes.count + 1
```

```
        case nil:
```

```
            fatalError("Unexpected CharacteristicTableViewSection raw value.")
```

```
}
```

```
}
```

```
/**
```

Looks up the appropriate service type for the row in the list and returns a localized version,

or 'None' if the row doesn't correspond to any valid service type.

- parameter row: The row to look up.

- returns: The localized service type in that row, or 'None'.

```
*/
```

```
func displayedServiceTypeForRow(_ row: Int) -> String {
```

```
    let serviceTypes = HMService.validAssociatedServiceTypes
```

```
    if row < serviceTypes.count {
```

```
        return HMService.localizedDescriptionForServiceType(serviceTypes[row])
```

```
    }
```

```

return NSLocalizedString("None", comment: "None")

}

```

```

/**

```

Evaluates whether or not a service type is selected for a given row.

- parameter row: The selected row.

- returns: true if the current row is a valid service type, false otherwise

```

*/

```

```

func serviceTypeIsSelectedForRow(_ row: Int) -> Bool {

```

```

    let serviceTypes = HMService.validAssociatedServiceTypes

```

```

    if row >= serviceTypes.count {

```

```

        return service.associatedServiceType == nil

```

```

    }

```

```

    if let associatedServiceType = service.associatedServiceType {

```

```

        return serviceTypes[row] == associatedServiceType

```

เอกสารนี้เป็นเอกสารที่สงวนลิขสิทธิ์ไว้สำหรับใช้ในการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า

ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

    }

    return false
}

/// Generates a cell for an associated service.

private func tableView(_ tableView: UITableView,
associatedServiceTypeCellForRowAtIndexPath indexPath: IndexPath) ->
UITableViewCell {

    let cell = tableView.dequeueReusableCell(withIdentifier:
Identifiers.serviceTypeCell, for:
indexPath)

    cell.textLabel?.text = displayedServiceTypeForRow(indexPath.row)

    cell.accessoryType = serviceTypeIsSelectedForRow(indexPath.row) ?
.checkmark : .none

    return cell
}

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

/**

Generates a characteristic cell based on the type of characteristic

located at the specified index path.

*/

private func tableView(_ tableView: UITableView,
characteristicCellForRowAtIndexPath
indexPath: IndexPath) -> UITableViewCell {

let characteristic = service.characteristics[indexPath.row]

var reuseIdentifier = Identifiers.characteristicCell

if (characteristic.isReadOnly || characteristic.isWriteOnly) &&
!allowsAllWrites {

reuseIdentifier = Identifiers.characteristicCell

}

else if characteristic.isBoolean {

reuseIdentifier = Identifiers.switchCharacteristicCell

```



```

    }

    else if characteristic.hasPredeterminedValueDescriptions {

        reuseIdentifier = Identifiers.segmentedControlCharacteristicCell

    }

//    else if characteristic.isNumeric {

//        reuseIdentifier = Identifiers.sliderCharacteristicCell

//    }

    else if characteristic.isTextWritable {

        reuseIdentifier = Identifiers.textCharacteristicCell

    }

    let cell = tableView.dequeueReusableCell(withIdentifier: reuseIdentifier, for:
indexPath) as!
CharacteristicCell

    cell.showsFavorites = showsFavorites

    cell.delegate = delegate

    cell.characteristic = characteristic

```

```

        return cell
    }

    /// Uses convenience methods to generate a cell based on the index path's section.

    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath)
    ->
    UITableViewCell {
        switch CharacteristicTableViewSection(rawValue: indexPath.section) {
        case .characteristics?:
            return self.tableView(tableView, characteristicCellForRowAtIndexPath:
indexPath)

        case .associatedServiceType?:
            return self.tableView(tableView,
associatedServiceTypeCellForRowAtIndexPath:
indexPath)

        case nil:

```

```

        fatalError("Unexpected CharacteristicTableViewSection raw value.")

    }

}

/// - returns: A localized string for the section.

func tableView(_ tableView: UITableView, titleForHeaderInSection section: Int)
-> String? {

    switch CharacteristicTableViewSection(rawValue: section) {

    case .characteristics?:

        return NSLocalizedString("Characteristics", comment: "Characteristics")

    case .associatedServiceType?:

        return NSLocalizedString("Associated Service Type", comment:
"Associated Service

Type")

    case nil:

        fatalError("Unexpected CharacteristicTableViewSection raw value.")

```

```

    }

    }

}

CharacteristicsTableViewDataSource

import UIKit

import HomeKit

/// A UITableViewCell subclass for displaying a service and the room and accessory
where it
resides.

class ServiceCell: UITableViewCell {

    // MARK: Properties

    var includeAccessoryText = true

    /// Required init.

    required init?(coder aDecoder: NSCoder) {

```

```

super.init(coder: aDecoder)

}

```

```

/**

```

The cell's service.

When the service is set, the cell's textLabel will contain the service's name or the accessory's name if the service has no name.

The detail text will contain information about where this service lives.

```

*/

```

```

var service: HMService? {

```

```

    didSet {

```

```

        if let service = service,

```

```

            let accessory = service.accessory {

```

```

                textLabel?.text = service.name

```

```

                let accessoryName = accessory.name

```

```

                let roomName = accessory.room!.name

```

```

                if includeAccessoryText {

```



```

        let inIdentifier = NSLocalizedString("%@ in %@", comment:
"Accessory in Room")

        detailTextLabel?.text = String(format: inIdentifier, accessoryName,
roomName)

    }

    else {

        detailTextLabel?.text = ""

    }

}

}

}

}

}

}

ServiceCell

import UIKit

import HomeKit

/**

```

A CharacteristicCell subclass that contains a text field.

Used for text-input characteristics.

*/

```
class TextCharacteristicCell: CharacteristicCell, UITextFieldDelegate {
```

```
    // MARK: Properties
```

```
    @IBOutlet weak var textField: UITextField!
```

```
    override var characteristic: HMCharacteristic! {
```

```
        didSet {
```

```
            textField.alpha = enabled ? 1.0 : CharacteristicCell.DisabledAlpha
```

```
            textField.isUserInteractionEnabled = enabled
```

```
        }
```

```
    }
```

```
    /// If notify is false, sets the text field's text from the value.
```

```
    override func setValue(_ newValue: CellValueType?, notify: Bool) {
```

```
        super.setValue(newValue, notify: notify)
```

```
        if !notify {
```

```

        if let newStringValue = newValue as? String {

            textField.text = newStringValue

        }

    }

}

/// Dismiss the keyboard when "Go" is clicked

func textFieldShouldReturn(_ textField: UITextField) -> Bool {

    textField.resignFirstResponder()

    return true

}

/// Sets the value of the characteristic when editing is complete.

func textFieldDidEndEditing(_ textField: UITextField) {

    setValue(textField.text as NSString?, notify: true)

}

}

```

TextCharacteristicCell

```
import UIKit
```

```
import HomeKit
```

```
/**
```

A CharacteristicCell subclass that contains a slider.

Used for numeric characteristics that have a continuous range of options.

```
*/
```

```
class SliderCharacteristicCell: CharacteristicCell {
```

```
// MARK: Properties
```

```
@IBOutlet weak var valueSlider: UISlider!
```

```
override var characteristic: HMCharacteristic! {
```

```
    didSet {
```

```
        valueSlider.alpha = enabled ? 1.0 : CharacteristicCell.DisabledAlpha
```

```
        valueSlider.isUserInteractionEnabled = enabled
```

```
    }
```

```

willSet(newCharacteristic) {

    // These are sane defaults in case the max and min are not set.

    valueSlider.minimumValue = newCharacteristic.metadata?.minimumValue
as? Float ?? 0.0

    valueSlider.maximumValue = newCharacteristic.metadata?.maximumValue
as? Float ??

100.0

}

}

/// If notify is false, sets the valueSlider's represented value.
override func setValue(_ newValue: CellValueType?, notify: Bool) {
    super.setValue(newValue, notify: notify)

    if let newValue = newValue as? NSNumber, !notify {

        valueSlider.value = newValue.floatValue

    }

}

```


/**

Restricts a value to the step value provided in the cell's

characteristic's metadata.

- parameter sliderValue: The provided value.

- returns: The value adjusted to align with a step value.

*/

```
func roundedValueForSliderValue(_ value: Float) -> Float {
```

```
    if let metadata = characteristic.metadata,
```

```
        let stepValue = metadata.stepValue as? Float
```

```
        , stepValue > 0 {
```

```
            let newStep = roundf(value / stepValue)
```

```
            let stepped = newStep * stepValue
```

```
            return stepped
```

```
    }
```

```
    return value
```

```

}

// Sliders don't update immediately, because sliders generate many updates.

override class var updatesImmediately: Bool {

    return false

}

/**
 Responds to a slider change and sets the cell's value.
 - parameter slider: The slider that changed.
 */

@IBAction func didChangeSliderValue(_ slider: UISlider) {

    let value = roundedValueForSliderValue(slider.value)

    setValue(value as NSNumber?, notify: true)

}

```

SliderCharacteristicCell

import UIKit

import HomeKit

typealias CellValueType = NSCopying

/// Methods for handling cell reads and updates.

protocol CharacteristicCellDelegate {

/**

Called whenever the control within the cell updates its value.

- parameter cell: The cell which has updated its value.
- parameter newValue: The new value represented by the cell's control.
- parameter characteristic: The characteristic the cell represents.
- parameter immediate: Whether or not to update external values immediately.

For example, Slider cells should not update immediately upon value change,

so their values are cached and updates are coalesced. Subclasses can decide whether or not their values are meant to be updated immediately.

*/

```
func characteristicCell(_ cell: CharacteristicCell, didUpdateValue value: CellValueType,
```

```
forCharacteristic characteristic: HMCharacteristic, immediate: Bool)
```

/**

Called when the characteristic cell needs to reload its value from an external source.

Consider using this call to look up values in memory or query them from an accessory.

- parameter cell: The cell requesting a value update.

- parameter characteristic: The characteristic for whose value the cell is asking.

- parameter completion: The closure that the cell provides to be called when values have

been read successfully.

*/

```

func characteristicCell(_ cell: CharacteristicCell,
readInitialValueForCharacteristic characteristic:
HMCharacteristic, completion: @escaping (CellValueType?, Error?) -> Void)
}

/**
 A UITableViewCell subclass that displays the current value of an
 HMCharacteristic and
 notifies its delegate of changes. Subclasses of this class will provide additional
 controls
 to display different kinds of data.
 */
class CharacteristicCell: UITableViewCell {

    /// An alpha percentage used when disabling cells.

    static let DisabledAlpha: CGFloat = 0.4

    /// Required init.

    required init?(coder aDecoder: NSCoder) {

        super.init(coder: aDecoder)

```



```
}
```

/// Subclasses can return false if they have many frequent updates that should be deferred.

```
class var updatesImmediately: Bool {
```

```
    return true
```

```
}
```

```
// MARK: Properties
```

```
@IBOutlet weak var typeLabel: UILabel!
```

```
@IBOutlet weak var valueLabel: UILabel!
```

```
@IBOutlet weak var favoriteButton: UIButton!
```

```
@IBOutlet weak var favoriteButtonWidthConstraint: NSLayoutConstraint!
```

```
@IBOutlet weak var favoriteButtonHeightContraint: NSLayoutConstraint!
```

```
/**
```

Show / hide the favoriteButton and adjust the constraints

to ensure proper layout.

*/

var showsFavorites = false {

didSet {

if showsFavorites {

favoriteButton.isHidden = false

favoriteButtonWidthConstraint.constant

favoriteButtonHeightConstraint.constant

}

else {

favoriteButton.isHidden = true

favoriteButtonWidthConstraint.constant = 15.0

}

}

}

/**

- returns: true if the represented characteristic is reachable;

false otherwise.

*/

var enabled: Bool {

return (characteristic.service?.accessory?.isReachable ?? false)

}

/**

The value currently represented by the cell.

This is not necessarily the value of this cell's characteristic,

because the cell's value changes independently of the characteristic.

*/

var value: AnyObject?

/// The delegate that will respond to cell value changes.

var delegate: CharacteristicCellDelegate?

```
/**
```

The characteristic represented by this cell.

When this is set, the cell populates based on

the characteristic's value and requests an initial value

from its delegate.

```
*/
```

```
var characteristic: HMCharacteristic! {
```

```
    didSet {
```

```
        typeLabel.text = characteristic.localizedCharacteristicType
```

```
        selectionStyle = characteristic.isIdentify ? .default : .none
```

```
        setValue(characteristic.value as? CellValueType, notify: false)
```

```
        if characteristic.isWriteOnly {
```

```
            // Don't read the value for write-only characteristics.
```

```
            return
```

```

    }

    // Set initial state of the favorite button

    // favoriteButton.isSelected = characteristic.isFavorite

    // "Enable" the cell if the accessory is reachable or we are displaying the
    favorites.

    // Configure the views.
    typeLabel.alpha = enabled ? 1.0 : CharacteristicCell.DisabledAlpha
    valueLabel?.alpha = enabled ? 1.0 : CharacteristicCell.DisabledAlpha

    if enabled {

        delegate?.characteristicCell(self, readInitialValueForCharacteristic:
characteristic) { value,

error in

        if let error = error {

            print("HomeKit: Error reading value for characteristic

```



```

    }
}
}
}
}

/// Resets the value label to the localized description
IMCharacteristic+Readability.

func resetValueLabel() {

```

```
else {
```

$$\}$$

}

}

```
func resetValueLabel() {
```

```
valueLabel?.text = characteristic.localizedDescriptionForValue(value)
```

$$\}$$
$$\}$$

```
/**
```

Toggles the star button and saves the favorite status
of the characteristic in the FavoriteManager.

```
*/
```

```
// @IBAction func didTapFavoriteButton(_ sender: UIButton) {  
//     sender.isSelected = !sender.isSelected  
//     characteristic.isFavorite = sender.isSelected  
// }
```

```
/**
```

Sets the cell's value and resets the label.

- parameter newValue: The new value.

- parameter notify: If true, the cell notifies its delegate of the change.

```
*/
```

```
func setValue(_ newValue: CellValueType?, notify: Bool) {
```

```
    value = newValue
```

```

if let newValue = newValue {

    resetValueLabel()

    /*

    We do not allow the setting of nil values from the app,

    but we do have to deal with incoming nil values.

    */

    if notify {
        delegate?.characteristicCell(self, didUpdateValue: newValue,
        forCharacteristic:
        characteristic, immediate: type(of: self).updatesImmediately)
    }
}

}

}

}

CharacteristicCell

import UIKit

import HomeKit

```

/**

A CharacteristicCell subclass that contains a single switch.

Used for Boolean characteristics.

*/

class SwitchCharacteristicCell: CharacteristicCell {

// MARK: Properties

@IBOutlet weak var valueSwitch: UISwitch!

override var characteristic: HMCharacteristic! {

didSet {

valueSwitch.alpha = enabled ? 1.0 : CharacteristicCell.DisabledAlpha

valueSwitch.isUserInteractionEnabled = enabled

}

}

/// If notify is false, sets the switch to the value.

override func setValue(_ newValue: CellValueType?, notify: Bool) {

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปเผยแพร่โดยไม่ขออนุญาต

ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```
super.setValue(newValue, notify: notify)
```

```
if !notify {
```

```
    if let boolValue = newValue as? Bool {
```

```
        valueSwitch.setOn(boolValue, animated: true)
```

```
    }
```

```
}
```

```
}
```

```
/**
```

Responds to the switch updating and sets the
value to the switch's value.

- parameter valueSwitch: The switch that updated.

```
*/
```

```
@IBAction func didChangeSwitchValue(_ valueSwitch: UISwitch) {
```

```
    setValue(valueSwitch.isOn as NSNumber?, notify: true)
```

```
}
```



```
}
```

```
SwitchCharacteristicCell
```

```
import UIKit
```

```
import HomeKit
```

```
/**
```

A CharacteristicCell subclass that contains a UISegmentedControl.

Used for HMCharacteristics which have associated, non-numeric values, like Lock Management

State.

```
*/
```

```
class SegmentedControlCharacteristicCell: CharacteristicCell {
```

```
    // MARK: Properties
```

```
    @IBOutlet weak var segmentedControl: UISegmentedControl!
```

/**

Calls the super class's didSet, and also computes a list

of possible values.

*/

override var characteristic: HMCharacteristic! {

didSet {

segmentedControl.alpha = enabled ? 1.0 : CharacteristicCell.DisabledAlpha

segmentedControl.isUserInteractionEnabled = enabled

if let values = characteristic.allPossibleValues as? [Int] {

possibleValues = values

}

}

}

/**

The possible values for this characteristic.

When this is set, adds localized descriptions to the segmented control.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า

ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

*/

var possibleValues = [Int]() {

    didSet {

        segmentedControl.removeAllSegments()

        for index in 0..

```

Responds to the segmented control's segment changing.

Sets the value and notifies its delegate.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

- parameter sender: The segmented control that changed.

*/

```
@IBAction func segmentedControlDidChange(_ sender: UISegmentedControl) {
```

```
    let value = possibleValues[sender.selectedSegmentIndex]
```

```
    setValue(value as NSNumber?, notify: true)
```

```
}
```

/**

If notify is false, then this is an external change,
so update the selected segment on the segmented control.

- parameter newValue: The new value.

- parameter notify: Whether or not to notify the delegate.

*/

```
override func setValue(_ newValue: CellValueType?, notify: Bool) {
```

```
    super.setValue(newValue, notify: notify)
```

```
    if !notify {
```

```

        resetSelectedIndex()

    }

}

/// Sets the segmented control based on the set value.

func resetSelectedIndex() {

    if let intValue = value as? Int, let index = possibleValues.firstIndex(of:
intValue) {
        segmentedControl.selectedSegmentIndex = index
    }
}

}

SegmentedControlCharacteristicCell

import UIKit

import HomeKit

/// Represents the sections in the ServicesViewController.

```



```

enum AccessoryTableViewSection: Int {

    case services, bridgedAccessories

}

/**

    A view controller which displays all the services of a provided accessory, and

    passes its cell delegate onto a CharacteristicsViewController.

*/
class ServicesViewController: HMCatalogViewController, HMAccessoryDelegate
{

    // MARK: Types

    struct Identifiers {

        static let accessoryCell = "AccessoryCell"

        static let serviceCell = "ServiceCell"

        static let showServiceSegue = "Show Service"

    }

```

```
// MARK: Properties
```

```
var accessory: HMAccessory!
```

```
lazy var cellDelegate: CharacteristicCellDelegate = AccessoryUpdateController()
```

```
var showsFavorites = false
```

```
var allowsAllWrites = false
```

```
var onlyShowsControlServices = false
```

```
var displayedServices = [HMService]()
```

```
var bridgedAccessories = [HMAccessory]()
```

```
// MARK: View Methods
```

```
/// Configures table view.
```

```
override func viewDidLoad() {
```

```
    super.viewDidLoad()
```

```
    tableView.estimatedRowHeight = 44.0
```

```
    tableView.rowHeight = UITableView.automaticDimension
```

```
}
```

```
/// Reloads the view.
```

```
override func viewWillAppear(_ animated: Bool) {
```

```
    super.viewWillAppear(animated)
```

```
    updateTitle()
```

```
    reloadData()
```

```
}
```

```
/// Pops the view controller, if required.
```

```
override func viewDidAppear(_ animated: Bool) {
```

```
    super.viewDidAppear(animated)
```

```
    if shouldPopViewController() {
```

```
        _ = navigationController?.popToRootViewController(animated: true)
```

```
    }
```

```
}
```

```
/**
```

Passes the CharacteristicsViewController the service from the cell and

configures the view controller.

*/

override func prepare(for segue: UIStoryboardSegue, sender: Any?) {

super.prepare(for: segue, sender: sender)

guard segue.identifier == Identifiers.showServiceSegue else { return }

if let indexPath = tableView.indexPath(for: sender as! UITableViewCell) {

let selectedService = displayedServices[indexPath.row]

let characteristicsViewController

segue.intendedDestinationViewController as!

CharacteristicsViewController

characteristicsViewController.showsFavorites = showsFavorites

characteristicsViewController.allowsAllWrites = allowsAllWrites

characteristicsViewController.service = selectedService

characteristicsViewController.cellDelegate = cellDelegate

}

}

```
/**
```

```
- returns: true if our accessory is no longer in the  
current home's list of accessories.
```

```
*/
```

```
private func shouldPopViewController() -> Bool {
```

```
for accessory in homeStore.home!.accessories {
```

```
if accessory == accessory {
```

```
return false
```

```
}
```

```
}
```

```
return true
```

```
}
```

```
// MARK: Delegate Registration
```

```
/**
```

```
Registers as the delegate for the current home
```



```

and for the current accessory.

*/

override func registerAsDelegate() {

    super.registerAsDelegate()

    accessory.delegate = self

}

// MARK: Table View Methods

/// Two sections if we're showing bridged accessories.

override func numberOfSections(in tableView: UITableView) -> Int {

    if accessory.uniqueIdentifiersForBridgedAccessories != nil {

        return 2

    }

    return 1

}

}

```

/**

Section 1 contains the services within the accessory.

Section 2 contains the bridged accessories.

*/

```
override func tableView(_ tableView: UITableView, numberOfRowsInSectionSection: Int) -> Int {
```

```
    switch AccessoryTableViewSection(rawValue: section) {
```

```
        case .services?:
```

```
            return displayedServices.count
```

```
        case .bridgedAccessories?:
```

```
            return bridgedAccessories.count
```

```
        case nil:
```

```
            fatalError("Unexpected AccessoryTableViewSection raw value.")
```

```
    }
```

```
}
```

```
/**
```

- returns: A Service or Bridged Accessory Cell based

on the section.

*/

override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->

UITableViewCell {

switch AccessoryTableViewSection(rawValue: indexPath.section) {

case .services?:

return self.tableView(tableView, serviceCellForRowAtIndexPath: indexPath)

case .bridgedAccessories?:

return self.tableView(tableView, bridgedAccessoryCellForRowAtIndexPath: indexPath)

case nil:

fatalError("Unexpected AccessoryTableViewSection raw value.")

}

}

/**

- returns: A cell containing the name of a bridged

accessory at a given index path.

*/

```
func tableView(_ tableView: UITableView,
bridgedAccessoryCellForRowAtIndexPath indexPath:
IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier:
Identifiers.accessoryCell, for:
indexPath)

    let accessory = bridgedAccessories[indexPath.row]
    cell.textLabel?.text = accessory.name

    return cell
}
```

/**

- returns: A cell containing the name of a service at

a given index path, as well as a localized

description of its service type.

*/

```
func tableView(_ tableView: UITableView, serviceCellForRowAtIndexPath  
indexPath: IndexPath)
```

```
-> UITableViewCell {
```

```
    let cell = tableView.dequeueReusableCell(withIdentifier:  
Identifiers.serviceCell, for:  
indexPath)
```

```
    let service = displayedServices[indexPath.row]
```

```
    // Inherit the name from the accessory if the Service doesn't have one.
```

```
    cell.textLabel?.text = service.name
```

```
    cell.detailTextLabel?.text = service.localizedDescription
```

```
    return cell
```

```
}
```

```

/// - returns: A title string for the section.

override func tableView(_ tableView: UITableView, titleForHeaderInSection
section: Int) ->

String? {

    switch AccessoryTableViewSection(rawValue: section) {

        case .services?:

            return NSLocalizedString("Services", comment: "Services")

        case .bridgedAccessories?:

            return NSLocalizedString("Bridged Accessories", comment: "Bridged
Accessories")

        case nil:

            fatalError("Unexpected AccessoryTableViewSection raw value.")

    }

}

}

/// - returns: A localized description of the accessories bridged status.

```



```

    override func tableView(_ tableView: UITableView, titleForFooterInSection
section: Int) ->

String? {

    if accessory.isBridged && AccessoryTableViewSection(rawValue: section)!
== .services {

        let formatString = NSLocalizedString("This accessory is being bridged into
HomeKit by

%@", comment: "Bridge Description")

        if let bridge = home.bridgeForAccessory(accessory) {

            return String(format: formatString, bridge.name)

        }

        else {

            return NSLocalizedString("This accessory is being bridged into
HomeKit.", comment:

"Bridge Description Without Bridge")

        }

    }

    return nil

}

```

```
// MARK: Helper Methods
```

```
/// Updates the navigation bar's title.
```

```
func updateTitle() {
```

```
    navigationItem.title = accessory.name
```

```
}
```

```
/**
```

Updates the title, resets the displayed services based on
view controller configurations, reloads the bridge accessory
array and reloads the table view.

```
*/
```

```
private func reloadData() {
```

```
    displayedServices = accessory.services.sortByLocalizedNames()
```

```
    if onlyShowsControlServices {
```

```
        // We are configured to only show control services, filter the array.
```

```
        displayedServices = displayedServices.filter { service -> Bool in
```

```

        return service.isControlType

    }

}

if let identifiers = accessory.uniqueIdentifiersForBridgedAccessories {

    bridgedAccessories =
    home.accessoriesWithIdentifiers(identifiers).sortByLocalizedName()

}

tableView.reloadData()

}

// MARK: HMAccessoryDelegate Methods

/// Reloads the title based on the accessories new name.

func accessoryDidUpdateName(_ accessory: HMAccessory) {

    updateTitle()

}

```

```

/// Reloads the cell for the specified service.

func accessory(_ accessory: HMAccessory, didUpdateNameFor service:
HMService) {

    if let index = displayedServices.firstIndex(of: service) {

        let path = IndexPath(row: index, section:
AccessoryTableViewSection.services.rawValue)

        tableView.reloadRows(at: [path], with: .automatic)

    }
}

/// Reloads the view.

func accessoryDidUpdateServices(_ accessory: HMAccessory) {

    reloadData()

}

/// If our accessory has become unreachable, go back the previous view.

func accessoryDidUpdateReachability(_ accessory: HMAccessory) {

    if self.accessory == accessory {

```

```

        _ = navigationController?.popViewController(animated: true)

    }

}

}

ServicesViewController

import UIKit

import HomeKit

/// Represents the sections in the ModifyAccessoryViewController.
enum AddAccessoryTableViewSection: Int {

    case name, rooms, identify

    static let count = 3

}

/// Contains a method for notifying the delegate that the accessory was saved.

protocol ModifyAccessoryDelegate {

```

```

func accessoryViewController(_ accessoryViewController:
ModifyAccessoryViewController,

didSaveAccessory accessory: HMAccessory)

}

/// A view controller that allows for renaming, reassigning, and identifying
accessories before and

after they've been added to a home.

class ModifyAccessoryViewController: HMCatalogViewController,
HMAccessoryDelegate {

// MARK: Types

struct Identifiers {

static let roomCell = "RoomCell"

}

// MARK: Properties

// Update this if the accessory failed in any way.

```



```
private var didEncounterError = false
```

```
private var selectedIndexPath: IndexPath?
```

```
private var selectedRoom: HMRoom!
```

```
@IBOutlet weak var nameField: UITextField!
```

```
private lazy var activityIndicator = UIActivityIndicatorView(style: .gray)
```

```
private let saveAccessoryGroup = DispatchGroup()
```

```
private var editingExistingAccessory = false
```

```
// Strong reference, because we will replace the button with an activity indicator.
```

```
@IBOutlet /* strong */ var addButton: UIBarButtonItem!
```

```
var delegate: ModifyAccessoryDelegate?
```

```
var rooms = [HMRoom]()
```

```
var accessory: HMAccessory!
```

```
// MARK: View Methods
```

```
/// Configures the table view and initializes view elements.
```

```
override func viewDidLoad() {
```

```
    super.viewDidLoad()
```

```
    tableView.estimatedRowHeight = 44.0
```

```
    tableView.rowHeight = UITableView.automaticDimension
```

```
    selectedRoom = accessory.room ?? home.roomForEntireHome()
```

```
    // If the accessory belongs to the home already, we are in 'edit' mode.
```

```
    editingExistingAccessory = accessoryHasBeenAddedToHome()
```

```
    if editingExistingAccessory {
```

```
        // Show 'save' instead of 'add.'
```

```
        addButton.title = NSLocalizedString("Save", comment: "Save")
```

```
    }
```

```
else {
```

```
/*
```

If we're not editing an existing accessory, then let the back

button show in the left.

```
*/
```

```
navigationItem.leftBarButtonItem = nil
```

```
}
```

```
// Put the accessory's name in the 'name' field.
```

```
resetNameField()
```

```
// Register a cell for the rooms.
```

```
tableView.register(UITableViewCell.self, forCellReuseIdentifier:  
Identifiers.roomCell)
```

```
}
```

```
/**
```

Registers as the delegate for the current home

and the accessory.

*/

override func registerAsDelegate() {

super.registerAsDelegate()

accessory.delegate = self

}

/// Replaces the activity indicator with the 'Add' or 'Save' button.

func hideActivityIndicator() {

activityIndicator.stopAnimating()

navigationItem.rightBarButtonItem = addButton

}

/// Temporarily replaces the 'Add' or 'Save' button with an activity indicator.

func showActivityIndicator() {

navigationItem.rightBarButtonItem = UIBarButtonItem(customView:
activityIndicator)

activityIndicator.startAnimating()

```
}
```

```
/**
```

Called whenever the user taps the 'add' button.

This method:

1. Adds the accessory to the home, if not already added.
2. Updates the accessory's name, if necessary.
3. Assigns the accessory to the selected room, if necessary.

```
*/
```

```
@IBAction func didTapAddButton() {
```

```
    let name = trimmedName
```

```
    showActivityIndicator()
```

```
    if editingExistingAccessory {
```

```
        home(home, assignAccessory: accessory, toRoom: selectedRoom)
```

```
        updateName(name, forAccessory: accessory)
```

```
    }
```

```

else {

    saveAccessoryGroup.enter()

    home.addAccessory(accessory) { error in

        if let error = error {

            self.hideActivityIndicator()

            self.displayError(error)

            self.didEncounterError = true
        }
        else {
            // Once it's successfully added to the home, add it to the room that's
            selected.
            self.home(self.home, assignAccessory:self.accessory, toRoom:
            self.selectedRoom)

            self.updateName(name, forAccessory: self.accessory)

        }

        self.saveAccessoryGroup.leave()

    }

}

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้


```

saveAccessoryGroup.notify(queue: DispatchQueue.main) {

    self.hideActivityIndicator()

    if !self.didEncounterError {

        self.dismiss(nil)

    }

}

}

}

/**
    Informs the delegate that the accessory has been saved, and
    dismisses the view controller.

*/

@IBAction func dismiss(_ sender: AnyObject?) {

    delegate?.accessoryViewController(self, didSaveAccessory: accessory)

    if editingExistingAccessory {

        presentingViewController?.dismiss(animated: true, completion: nil)

    }

```

```

else {

    _ = navigationController?.popViewController(animated: true)

}

}

```

```
/**
```

- returns: true if the accessory has already been added to the home; false otherwise.

```
*/
```

```

func accessoryHasBeenAddedToHome() -> Bool {
    return home.accessories.contains(accessory)
}

```

```
/**
```

Updates the accessories name. This function will enter and leave the saved dispatch group.

If the accessory's name is already equal to the passed-in name, this method does nothing.

- parameter name: The new name for the accessory.

- parameter accessory: The accessory to rename.

*/

```
func updateName(_ name: String, forAccessory accessory: HMAccessory) {
```

```
    if accessory.name == name {
```

```
        return
```

```
    }
```

```
    saveAccessoryGroup.enter()
```

```
    accessory.updateName(name) { error in
```

```
        if let error = error {
```

```
            self.displayError(error)
```

```
            self.didEncounterError = true
```

```
        }
```

```
        self.saveAccessoryGroup.leave()
```

```
    }
```

```
}
```

/**

Assigns the given accessory to the provided room. This method will enter and leave the saved

dispatch group.

- parameter home: The home to assign.

- parameter accessory: The accessory to be assigned.

- parameter room: The room to which to assign the accessory.

*/

func home(_ home: HMHome, assignAccessory accessory: HMAccessory, toRoom room:

HMRoom) {

if accessory.room == room {

return

}

saveAccessoryGroup.enter()

home.assignAccessory(accessory, to: room) { error in

if let error = error {

```

        self.displayError(error)

        self.didEncounterError = true

    }

    self.saveAccessoryGroup.leave()

}

}

/// Tells the current accessory to identify itself.
func identifyAccessory() {
    accessory.identify { error in
        if let error = error {
            self.displayError(error)
        }
    }
}

}

}

/// Enables the name field if the accessory's name changes.

func resetNameField() {

```

```

var action: String

if editingExistingAccessory {

    action = NSLocalizedString("Edit %@", comment: "Edit Accessory")

}

else {

    action = NSLocalizedString("Add %@", comment: "Add Accessory")

}

navigationItem.title = String(format: action, accessory.name) as String
nameField.text = accessory.name
nameField.isEnabled = home.isAdmin
enableAddButtonIfApplicable()

}

/// Enables the save button if the name field is not empty.

func enableAddButtonIfApplicable() {

    addButton.isEnabled = home.isAdmin && trimmedName.count > 0

}

```



```

    /// - returns: The nameField's text, trimmed of newline and whitespace characters.

    var trimmedName: String {

        return nameField.text!.trimmingCharacters(in:
CharacterSet.whitespacesAndNewlines)

    }

    /// Enables or disables the add button.

    @IBAction func nameFieldDidChange(_ sender: AnyObject) {
        enableAddButtonIfApplicable()
    }

    // MARK: Table View Methods

    /// - returns: The number of AddAccessoryTableViewSections.

    override func numberOfSections(in tableView: UITableView) -> Int {

        return AddAccessoryTableViewSection.count

    }

```

```

/// - returns: The number rows for the rooms section. All other sections are static.

override func tableView(_ tableView: UITableView, numberOfRowsInSectionSection
section: Int) -> Int {

    switch AddAccessoryTableViewSection(rawValue: section) {

        case .rooms?:

            return home.allRooms.count

        case nil:

            fatalError("Unexpected AddAccessoryTableViewSection raw value.")

        default:

            return super.tableView(tableView, numberOfRowsInSection: section)

    }

}

/// - returns: UITableViewAutomaticDimension for dynamic cell, super otherwise.

override func tableView(_ tableView: UITableView, heightForRowAt indexPath:
IndexPath) ->

```

```

CGFloat {

    switch AddAccessoryTableViewSection(rawValue: indexPath.section) {

        case .rooms?:

            return UITableView.automaticDimension

        case nil:

            fatalError("Unexpected AddAccessoryTableViewSection raw value.")

        default:

            return super.tableView(tableView, heightForRowAt: indexPath)

    }

}

/// - returns: A 'room cell' for the rooms section, super otherwise.

override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) ->

UITableViewCell {

    switch AddAccessoryTableViewSection(rawValue: indexPath.section) {

```

```

        case .rooms?:

            return self.tableView(tableView, roomCellForRowAtIndexPath:
indexPath)

        case nil:

            fatalError("Unexpected AddAccessoryTableViewSection raw value.")

        default:

            return super.tableView(tableView, cellForRowAt: indexPath)
    }
}

/**

Creates a cell with the name of each room within the home, displaying a
checkmark if the

room

is the currently selected room.

*/

```

```

func tableView(_ tableView: UITableView, cellForRowAtIndexPath
indexPath: IndexPath)

> UITableViewCell {

    let cell = tableView.dequeueReusableCell(withIdentifier: Identifiers.roomCell,
for: indexPath)

    let room = home.allRooms[indexPath.row] as HMRoom

    cell.textLabel?.text = home.nameForRoom(room)

    // Put a checkmark on the selected room.
    cell.accessoryType = room == selectedRoom ? .checkmark : .none

    if !home.isAdmin {
        cell.selectionStyle = .none
    }

    return cell
}

```

```

/// Handles row selection based on the section.

override func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {

    tableView.deselectRow(at: indexPath, animated: true)

    switch AddAccessoryTableViewSection(rawValue: indexPath.section) {

    case .rooms?:

        guard home.isAdmin else { return }

        selectedRoom = home.allRooms[indexPath.row]

        let sections = IndexSet(integer:
AddAccessoryTableViewSection.rooms.rawValue)

        tableView.reloadSections(sections, with: .automatic)

    case .identify?:

        identifyAccessory()

```



```

case nil:

    fatalError("Unexpected AddAccessoryTableViewSection raw value.")

default: break

}

}

/// Required override.
override func tableView(_ tableView: UITableView, indentationLevelForRowAt
indexPath:
IndexPath) -> Int {

    return super.tableView(tableView, indentationLevelForRowAt:
IndexPath(row: 0, section:
indexPath.section))

}

// MARK: HMHomeDelegate Methods

```

```
// All home changes reload the view.
```

```
func home(_ home: HMHome, didUpdateNameFor room: HMRoom) {
```

```
    tableView.reloadData()
```

```
}
```

```
func home(_ home: HMHome, didAddRoom room: HMRoom) {
```

```
    tableView.reloadData()
```

```
}
```

```
func home(_ home: HMHome, didRemoveRoom room: HMRoom) {
```

```
    if selectedRoom == room {
```

```
        // Reset the selected room if ours was deleted.
```

```
        selectedRoom = homeStore.home!.roomForEntireHome()
```

```
    }
```

```
    tableView.reloadData()
```

```
}
```

```

func home(_ home: HMHome, didAddAccessory accessory: HMAccessory) {

    /*

    Bridged accessories don't call the original completion handler if their

    bridges are added to the home. We must respond to HMHomeDelegate's

    home(_:didAddAccessory:) and assign bridged accessories properly.

    */

    if selectedRoom != nil {

        self.home(home, assignAccessory: accessory, toRoom: selectedRoom)

    }

}

func home(_ home: HMHome, didUnblockAccessory accessory: HMAccessory)

{

    tableView.reloadData()

}

// MARK: HMAccessoryDelegate Methods

```

```

/// If the accessory's name changes, we update the name field.

func accessoryDidUpdateName(_ accessory: HMAccessory) {

    resetNameField()

}

}

ModifyAccessoryViewController

import HomeKit

/// An object that responds to CharacteristicCell updates and notifies HomeKit of
changes.

class AccessoryUpdateController: NSObject, CharacteristicCellDelegate {

    // MARK: Properties

    let updateQueue = DispatchQueue(label:

"com.sample.HMCatalog.CharacteristicUpdateQueue", attributes: [])

    lazy var pendingWrites = [HMCharacteristic:CellValueType]()

```

```

lazy var sentWrites = [HMCharacteristic:CellValueType]()

// Implicitly unwrapped optional because we need self to initialize.

var updateValueTimer: Timer!

/// Starts the update timer on creation.

override init() {
    super.init()
    startListeningForCellUpdates()
}

/// Responds to a cell change, and if the update was marked immediate, updates
the
characteristics.

func characteristicCell(_ cell: CharacteristicCell, didUpdateValue value:
CellValueType,

forCharacteristic characteristic: HMCharacteristic, immediate: Bool) {

    pendingWrites[characteristic] = value

```

```

    if immediate {

        updateCharacteristics()

    }

}

/**
    Reads the characteristic's value and calls the completion with the characteristic's
    value.

    If there is a pending write request on the same characteristic, the read is ignored
    to prevent
    "UI glitching".

*/

func characteristicCell(_ cell: CharacteristicCell,
readInitialValueForCharacteristic characteristic:
HMCharacteristic, completion: @escaping (CellValueType?, Error?) -> Void) {

    characteristic.readValue { error in

        self.updateQueue.sync {

            if let sentValue = self.sentWrites[characteristic] {

```



```

        completion(sentValue, nil)

    return

}

```

```

DispatchQueue.main.async {

    completion(characteristic.value as? CellValueType, error)

}

}

}

}

}

/// Creates and starts the update value timer.

func startListeningForCellUpdates() {

    updateValueTimer = Timer.scheduledTimer(timeInterval: 0.1, target: self,
selector:

#selector(self.updateCharacteristics), userInfo: nil, repeats: true)

}

```

```

/// Invalidates the update timer.

func stopListeningForCellUpdates() {

    updateValueTimer.invalidate()

}

/// Sends all pending requests in the array.

@objc func updateCharacteristics() {

    updateQueue.sync {

        for (characteristic, value) in self.pendingWrites {

            self.sentWrites[characteristic] = value

            characteristic.writeValue(value) { error in

                if let error = error {

                    print("HomeKit: Could not change value:
\\(error.localizedDescription).")

                }

                self.didCompleteWrite(characteristic, value: value)

```

```

    }

    }

    self.pendingWrites.removeAll()

    }

    }

    /**
     Synchronously adds the characteristic-value pair into the sentWrites map.
     - parameter characteristic: The HMCharacteristic to add.
     - parameter value: The value of the characteristic.

    */

    func didSendWrite(_ characteristic: HMCharacteristic, value: CellValueType) {

        updateQueue.sync {

            self.sentWrites[characteristic] = value

        }

    }

```

```
/**
```

Synchronously removes the characteristic-value pair from the sentWrites map.

- parameter characteristic: The HMCharacteristic to remove.

- parameter value: The value of the characteristic (unused, but included for clarity).

```
*/
```

```
func didCompleteWrite(_ characteristic: HMCharacteristic, value: AnyObject) {
```

```
    updateQueue.sync {
```

```
        _ = self.sentWrites.removeValue(forKey: characteristic)
```

```
    }
```

```
}
```

```
}
```

AccessoryUpdateController

```
import UIKit
```

```
import HomeKit
```

```
/// A view controller that lists rooms within a home and allows the user to add the  
rooms to a
```

```
provided zone.
```

```
class AddRoomViewController: HMCatalogViewController {
```

```
    // MARK: Types
```

```
    struct Identifiers {
```

```
        static let roomCell = "RoomCell"
```

```
    }
```

```
    // MARK: Properties
```

```
    var homeZone: HMZone!
```

```
    lazy var displayedRooms = [HMRoom]()
```

```
    lazy var selectedRooms = [HMRoom]()
```

```
    // MARK: View Methods
```

```

override func viewWillAppear(_ animated: Bool) {

    super.viewWillAppear(animated)

    title = homeZone.name

    resetDisplayedRooms()

}

/// Adds the selected rooms to the zone and dismisses the view.
@IBAction func dismiss(_ sender: AnyObject) {
    addSelectedRoomsToZoneWithCompletionHandler {
        self.dismiss(animated: true, completion: nil)
    }
}

```

/**

Creates a dispatch group, adds all of the rooms to the zone,

and runs the provided completion once all rooms have been added.

- parameter completion: A closure to call once all rooms have been added.

*/

```
func      addSelectedRoomsToZoneWithCompletionHandler(_      completion:
@escaping () -> Void) {
```

```
    let group = DispatchGroup()
```

```
    for room in selectedRooms {
```

```
        group.enter()
```

```
        homeZone.addRoom(room) { error in
```

```
            if let error = error {
```

```
                self.displayError(error)
```

```
            }
```

```
            group.leave()
```

```
        }
```

```
    }
```

```
    group.notify(queue: DispatchQueue.main, execute: completion)
```

```
}
```

```
// MARK: Table View Methods
```

/// - returns: The number of displayed rooms.

```
override func tableView(_ tableView: UITableView, numberOfRowsInSectionSection: Int) -> Int {
```

```
    return displayedRooms.count
```

```
}
```

/// - returns: A cell that includes the name of a room and a checkmark if it's intended to be added to the zone.

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
```

```
    UITableViewCell {
```

```
        let cell = tableView.dequeueReusableCell(withIdentifier: Identifiers.roomCell, for: indexPath)
```

```
        let room = displayedRooms[indexPath.row]
```

```
        cell.textLabel?.text = room.name
```

```
cell.accessoryType = selectedRooms.contains(room) ? .checkmark : .none
```

```
return cell
```

```
}
```

```
/// Adds the selected room to the selected rooms array and reloads that cell
```

```
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
```

```
    let room = displayedRooms[indexPath.row]
```

```
    if let index = selectedRooms.firstIndex(of: room) {
```

```
        selectedRooms.remove(at: index)
```

```
    }
```

```
    else {
```

```
        selectedRooms.append(room)
```

```
    }
```

```

tableView.reloadRows(at: [indexPath], with: .automatic)

}

/// Resets the list of displayed rooms and reloads the table.

func resetDisplayedRooms() {

    displayedRooms = home.roomsNotAlreadyInZone(homeZone,
includingRooms:
selectedRooms)

    if displayedRooms.isEmpty {
        self.dismiss(animated: true, completion: nil)
    }
    else {
        tableView.reloadData()
    }
}

}

// MARK: HMHomeDelegate Methods

```

```

/// If our zone was removed, dismiss this view.

func home(_ home: HMHome, didRemoveZone zone: HMZone) {

    if zone == homeZone {

        self.dismiss(animated: true, completion: nil)

    }

}

/// If our zone was renamed, reset our title.

func home(_ home: HMHome, didUpdateNameForZone zone: HMZone) {

    if zone == homeZone {

        title = zone.name

    }

}

// All home updates reset the displayed homes and reload the view.

func home(_ home: HMHome, didUpdateNameForRoom room: HMRoom) {

    resetDisplayedRooms()

```

```

}

func home(_ home: HMHome, didAddRoom room: HMRoom) {

    resetDisplayedRooms()

}

func home(_ home: HMHome, didRemoveRoom room: HMRoom) {

    resetDisplayedRooms()

}

func home(_ home: HMHome, didAddRoom room: HMRoom, toZone zone:
HMZone) {

    resetDisplayedRooms()

}

func home(_ home: HMHome, didRemoveRoom room: HMRoom, fromZone
zone: HMZone) {

    resetDisplayedRooms()

}

```



```

    }

}

AddRoomViewController

import UIKit

import HomeKit

/// A view controller that lists the rooms within a provided zone.
class ZoneViewController: HMCatalogViewController {

    // MARK: Types

    struct Identifiers {

        static let roomCell = "RoomCell"

        static let addCell = "AddCell"

        static let disabledAddCell = "DisabledAddCell"

        static let addRoomsSegue = "Add Rooms"

    }

}

```

// MARK: Properties

```
var homeZone: HMZone!
```

```
var rooms = [HMRoom]()
```

```
// MARK: View Methods
```

```
/// Reload the data and configure the view.
```

```
override func viewWillAppear(_ animated: Bool) {
```

```
    super.viewWillAppear(animated)
```

```
    title = homeZone.name
```

```
    reloadData()
```

```
}
```

```
/// If our data is invalid, pop the view controller.
```

```
override func viewDidAppear(_ animated: Bool) {
```

```
    super.viewDidAppear(animated)
```

```
    if shouldPopViewController() {
```

```
        _ = navigationController?.popViewController(animated: true)
```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า

ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 232 ไปถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

    }

}

/// Provide the zone to AddRoomViewController.

override func prepare(for segue: UIStoryboardSegue, sender: Any?) {

    super.prepare(for: segue, sender: sender)

    if segue.identifier == Identifiers.addRoomsSegue {

        let addViewController = segue.intendedDestinationViewController as!
AddRoomViewController
        addViewController.homeZone = homeZone
    }

}

// MARK: Helper Methods

/// Resets the internal list of rooms and reloads the table view.

private func reloadData() {

    rooms = homeZone.rooms.sortByLocalizedNames()

```

```

tableView.reloadData()

}

/// Sorts the internal list of rooms by localized name.

private func sortRooms() {

    rooms = rooms.sortByLocalizedNames()

}

/// - returns: The NSIndexPath where the 'Add Cell' should be located.
private var addIndexPath: NSIndexPath {

    return NSIndexPath(row: rooms.count, section: 0)

}

/**

- parameter indexPath: The index path in question.

- returns: true if the indexPath should contain
an 'add' cell, false otherwise

```

```
*/
```

```
private func indexPathIsAdd(_ indexPath: IndexPath) -> Bool {
```

```
    return indexPath.row == addIndexPath.row
```

```
}
```

```
/**
```

Reloads the addIndexPath.

This is typically used when something has changed to allow the user to add a room.

```
*/
```

```
private func reloadAddIndexPath() {
```

```
    tableView.reloadRows(at: [addIndexPath], with: .automatic)
```

```
}
```

```
/**
```

Adds a room to the internal array of rooms and inserts new row

into the table view.

- parameter room: The new HMRoom to add.

*/

```
private func didAddRoom(_ room: HMRoom) {
```

```
    rooms.append(room)
```

```
    sortRooms()
```

```
    if let newRoomIndex = rooms.firstIndex(of: room) {
```

```
        let newRoomIndexPath = IndexPath(row: newRoomIndex, section: 0)
```

```
        tableView.insertRows(at: [newRoomIndexPath], with: .automatic)
```

```
    }
```

```
    reloadAddIndexPath()
```

```
}
```

/**

Removes a room from the internal array of rooms and deletes

the row from the table view.

- parameter room: The HMRoom to remove.

*/

```
private func didRemoveRoom(_ room: HMRoom) {
```

```
    if let roomIndex = rooms.firstIndex(of: room) {
```

```
        rooms.remove(at: roomIndex)
```

```
        let roomIndexPath = IndexPath(row: roomIndex, section: 0)
```

```
        tableView.deleteRows(at: [roomIndexPath], with: .automatic)
```

```
    }
```

```
    reloadAddIndexPath()
```

```
}
```

/**

Reloads the cell corresponding a given room.

- parameter room: The HMRoom to reload.

*/

```
private func didUpdateRoom(_ room: HMRoom) {
```

```
    if let roomIndex = rooms.firstIndex(of: room) {
```

```
        let roomIndexPath = IndexPath(row: roomIndex, section: 0)
```

```
        tableView.reloadRows(at: [roomIndexPath], with: .automatic)
```

```
    }
```

```
}
```

/**

Removes a room from HomeKit and updates the view.

- parameter room: The HMRoom to remove.

*/

```
private func removeRoom(_ room: HMRoom) {
```

```
    didRemoveRoom(room)
```

```
    homeZone.removeRoom(room) { error in
```

```
        if let error = error {
```

```
            self.displayError(error)
```

```

        self.didAddRoom(room)

    }

}

}

/**
 - returns: true if our current home no longer
           exists, false otherwise.
 */
private func shouldPopViewController() -> Bool {
    for zone in home.zones {
        if zone == homeZone {
            return false
        }
    }

    return true
}

```

```
/**
```

```
- returns: true if more rooms can be added to this zone;
```

```
false otherwise.
```

```
*/
```

```
private var canAddRoom: Bool {
```

```
    return rooms.count < home.rooms.count
```

```
}
```

```
// MARK: Table View Methods
```

```
/// - returns: The number of rooms in the zone, plus 1 for the 'add' row.
```

```
override func tableView(_ tableView: UITableView, numberOfRowsInSectionSection: Int) -> Int {
```

```
    return rooms.count + 1
```

```
}
```

```
/// - returns: A cell containing the name of an HMRoom.
```

```

    override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) ->

UITableViewCell {

    if indexPathIsAdd(indexPath) {

        let reuseIdentifier = home.isAdmin && canAddRoom ? Identifiers.addCell :
Identifiers.disabledAddCell

        return tableView.dequeueReusableCell(withIdentifier: reuseIdentifier, for:
indexPath)
    }

    let cell = tableView.dequeueReusableCell(withIdentifier: Identifiers.roomCell,
for: indexPath)

    cell.textLabel?.text = rooms[indexPath.row].name

    return cell

}

```

```

/**

- returns: true if the cell is anything but an 'add' cell;

false otherwise.

*/

override func tableView(_ tableView: UITableView, canEditRowAt indexPath:
IndexPath) -> Bool
{
return home.isAdmin && !indexPathIsAdd(indexPath)
}

/// Deletes the room at the provided index path.

override func tableView(_ tableView: UITableView, commit editingStyle:
UITableViewCellEditingStyle, forRowAt indexPath: IndexPath) {

if editingStyle == .delete {

let room = rooms[indexPath.row]

removeRoom(room)

}
}

```



```
}
```

```
// MARK: HMHomeDelegate Methods
```

```
/// If our zone was removed, pop the view controller.
```

```
func home(_ home: HMHome, didRemoveZone zone: HMZone) {
```

```
    if zone == homeZone {
```

```
        _ = navigationController?.popViewController(animated: true)
```

```
    }
```

```
}
```

```
/// If our zone was renamed, update the title.
```

```
func home(_ home: HMHome, didUpdateNameForZone zone: HMZone) {
```

```
    if zone == homeZone {
```

```
        title = zone.name
```

```
    }
```

```
}
```

```
/// Update the row for the room.
```

```
func home(_ home: HMHome, didUpdateNameForRoom room: HMRoom) {
```

```
    didUpdateRoom(room)
```

```
}
```

```
/**
```

A room has been added, we may be able to add it to the zone.

Reload the 'addIndexPath'

```
*/
```

```
func home(_ home: HMHome, didAddRoom room: HMRoom) {
```

```
    reloadAddIndexPath()
```

```
}
```

```
/**
```

A room has been removed, attempt to remove it from the room.

This will always reload the 'addIndexPath'.

```
*/
```

```
func home(_ home: HMHome, didRemoveRoom room: HMRoom) {
```

เอกสารนี้เป็นเอกสารสงวนลิขสิทธิ์สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปเผยแพร่โดยไม่ขออนุญาต

ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 244 ไปถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

        didRemoveRoom(room)

    }

    /// If the room was added to our zone, add it to the view.

    func home(_ home: HMHome, didAddRoom room: HMRoom, toZone zone:
HMZone) {

        if zone == homeZone {

            didAddRoom(room)

        }

    }

    /// If the room was removed from our zone, remove it from the view.

    func home(_ home: HMHome, didRemoveRoom room: HMRoom, fromZone
zone: HMZone) {

        if zone == homeZone {

            didRemoveRoom(room)

        }

    }
}

```

```
}
```

```
ZoneViewController
```

```
import UIKit
```

```
import HomeKit
```

```
// Represents the sections in the HomeListConfigurationViewController.
```

```
enum HomeListSection: Int {
```

```
    case homes, primaryHome
```

```
    static let count = 2
```

```
}
```

```
/**
```

A HomeListViewController subclass which allows the user to add and remove

homes and set the primary home.

```
*/
```

```
class HomeListConfigurationViewController: HomeListViewController {
```

```
    // MARK: Types
```

```

struct Identifiers {

    static let addHomeCell = "AddHomeCell"

    static let noHomesCell = "NoHomesCell"

    static let primaryHomeCell = "PrimaryHomeCell"

    static let homeCell = "HomeCell"

}

override func viewDidLoad() {

    navigationItem.title = "Timer"

    self.tableView.contentInset = UIEdgeInsets(top: 44,left: 0,bottom: 0,right: 0);

    // Do any additional setup after loading the view.

}

```

// MARK: Table View Methods

```

/// - returns: The number of sections in the HomeListSection enum.

override func numberOfSections(in tableView: UITableView) -> Int {

    return HomeListSection.count

}

/// Provides the number of rows in the section using the internal home's list.

override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {

    switch HomeListSection(rawValue: section) {

        // Add row.

        case .homes?:

            return homes.count + 1

        // 'No homes' row.

        case .primaryHome?:

            return max(homes.count, 1)
    }
}

```



```

        case nil: fatalError("Unexpected HomeListSection raw value.")

    }

}

/**
Generates and configures either a content cell or an add cell using the
provided index path.
*/
override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) ->
UITableViewCell {

    if indexPathIsAdd(indexPath) {

        return tableView.dequeueReusableCell(withIdentifier:
Identifiers.addHomeCell, for:
indexPath)

    }

    else if homes.isEmpty {

        return tableView.dequeueReusableCell(withIdentifier:
Identifiers.noHomesCell, for:

```

```
indexPath)
```

```
}
```

```
let reuseIdentifier: String
```

```
switch HomeListSection(rawValue: indexPath.section) {
```

```
case .homes?:
```

```
    reuseIdentifier = Identifiers.homeCell
```

```
case .primaryHome?:
```

```
    reuseIdentifier = Identifiers.primaryHomeCell
```

```
case nil: fatalError("Unexpected HomeListSection raw value.")
```

```
}
```

```
let cell = tableView.dequeueReusableCell(withIdentifier: reuseIdentifier, for:  
indexPath)
```

```
let home = homes[indexPath.row]
```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```
cell.textLabel!.text = home.name
```

```
cell.detailTextLabel?.text = sharedTextForHome(home)
```

```
// Mark the primary home with checkmark.
```

```
if HomeListSection(rawValue: indexPath.section) == .primaryHome {
```

```
    if home == homeManager.primaryHome {
```

```
        cell.accessoryType = .checkmark
```

```
    }
```

```
    else {
```

```
        cell.accessoryType = .none
```

```
    }
```

```
}
```

```
return cell
```

```
}
```

```
/// Homes in the list section can be deleted. The add row cannot be deleted.
```

เอกสารนี้เป็นเอกสารสงวนลิขสิทธิ์สำหรับการใช้งานเพื่อการศึกษาเท่านั้น เมื่อเผยแพร่ไปใช้ประโยชน์ด้านการค้า

ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 251 ไปถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

    override func tableView(_ tableView: UITableView, canEditRowAt indexPath:
IndexPath) -> Bool

{

    return HomeListSection(rawValue: indexPath.section) == .homes &&

!indexPathIsAdd(indexPath)

}

/// Only the 'primary home' section has a title.

    override func tableView(_ tableView: UITableView, titleForHeaderInSection
section: Int) ->
String? {

        if HomeListSection(rawValue: section) == .primaryHome {

            return NSLocalizedString("Primary Home", comment: "Primary Home")

        }

        return nil

    }
}

```

```

/// Provides subtext about the use of designating a "primary home".

override func tableView(_ tableView: UITableView, titleForFooterInSection
section: Int) ->

String? {

    if section == HomeListSection.primaryHome.rawValue {

        return NSLocalizedString("The primary home is used by Siri to route
commands if the home
is not specified.", comment: "Primary Home Description")
    }

    return nil
}

/**

If selecting a regular home, a segue will be performed.

If this method is called, the user either selected the 'add' row,

a primary home cell, or the No Homes cell.

*/

override func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {

```

```

tableView.deselectRow(at: indexPath, animated: true)

if indexPathIsAdd(indexPath) {

    addNewHome()

}

else if indexPathIsNone(indexPath) {

    return

}

else if HomeListSection(rawValue: indexPath.section) == .primaryHome {
    let newPrimaryHome = homes[indexPath.row]
    updatePrimaryHome(newPrimaryHome)
}

}

/// Removes the home from HomeKit if the row is deleted.

override func tableView(_ tableView: UITableView, commit editingStyle:
UITableViewCell.EditingStyle, forRowAt indexPath: IndexPath) {

    if editingStyle == .delete {

        removeHomeAtIndexPath(indexPath)

```



```

    }

}

// MARK: Helper Methods

/**
    Updates the primary home in HomeKit and reloads the view.

    If the home is already selected, no action is taken.

    - parameter newPrimaryHome: The new HMHome to set as the primary home.
*/
private func updatePrimaryHome(_ newPrimaryHome: HMHome) {

    guard newPrimaryHome != homeManager.primaryHome else { return }

    homeManager.updatePrimaryHome(newPrimaryHome) { error in

        if let error = error {

            self.displayError(error)

            return

```

```

    }

    self.didUpdatePrimaryHome()

}

}

/// Reloads the 'primary home' section.
private func didUpdatePrimaryHome() {
    let primaryIndexSet = IndexSet(integer:
HomeListSection.primaryHome.rawValue)

    tableView.reloadSections(primaryIndexSet, with: .automatic)
}

/**

Removed the home at the specified index path from HomeKit and updates the
view.

```

- parameter indexPath: The NSIndexPath of the home to remove.

*/

```
private func removeHomeAtIndexPath(_ indexPath: IndexPath) {
```

```
    let home = homes[indexPath.row]
```

```
    // Remove the home from the data structure. If it fails, put it back.
```

```
    didRemoveHome(home)
```

```
    homeManager.removeHome(home) { error in
```

```
        if let error = error {
```

```
            self.displayError(error)
```

```
            self.didAddHome(home)
```

```
            return
```

```
        }
```

```
    }
```

```
}
```

/**

Presents an alert controller so the user can provide a name. If committed,

the home is created.

*/

private func addNewHome() {

let attributedType = NSAttributedString("Home", comment: "Home")

let placeholder = NSAttributedString("Apartment", comment: "Apartment")

presentAddAlertWithAttributeType(attributedType, placeholder: placeholder)

{ name in

self.addHomeWithName(name)

}

}

/**

Removes a home from the internal structure and updates the view.

- parameter home: The HMHome to remove.

*/

override func didRemoveHome(_ home: HMHome) {

```

guard let index = homes.firstIndex(of: home) else { return }

let indexPath = IndexPath(row: index, section:
HomeListSection.homes.rawValue)

homes.remove(at: index)

let primaryIndexPath = IndexPath(row: index, section:
HomeListSection.primaryHome.rawValue)

/*
If there aren't any homes, we still want one cell to display 'No Homes'.
Just reload.
*/

tableView.beginUpdates()

if homes.isEmpty {

    tableView.reloadRows(at: [primaryIndexPath], with: .fade)

}

else {

    tableView.deleteRows(at: [primaryIndexPath], with: .automatic)

```

```

    }

    tableView.deleteRows(at: [indexPath], with: .automatic)

    tableView.endUpdates()

}

/// Adds the home to the internal structure and updates the view.
override func didAddHome(_ home: HMHome) {
    homes.append(home)
    sortHomes()
    guard let newHomeIndex = homes.firstIndex(of: home) else { return }

    let indexPath = IndexPath(row: newHomeIndex, section:
HomeListSection.homes.rawValue)

    let primaryIndexPath = IndexPath(row: newHomeIndex, section:
HomeListSection.primaryHome.rawValue)

```



```

tableView.beginUpdates()

if homes.count == 1 {

    tableView.reloadRows(at: [primaryIndexPath], with: .fade)

}

else {

    tableView.insertRows(at: [primaryIndexPath], with: .automatic)
}

tableView.insertRows(at: [indexPath], with: .automatic)
tableView.endUpdates()

}

/**

Creates a new home with the provided name, adds the home to HomeKit

and reloads the view.

*/

```

```

private func addHomeWithName(_ name: String) {

```

```

homeManager.addHome(withName: name) { newHome, error in

    if let error = error {

        self.displayError(error)

        return

    }

    self.didAddHome(newHome!)
}
}

/// - returns: true if the index path is the 'add row'; false otherwise.

private func indexPathIsAdd(_ indexPath: IndexPath) -> Bool {

    return HomeListSection(rawValue: indexPath.section) == .homes &&

        indexPath.row == homes.count

}

```

```

/// - returns: true if the index path is the 'No Homes' cell; false otherwise.

private func indexPathIsNone(_ indexPath: IndexPath) -> Bool {

    return HomeListSection(rawValue: indexPath.section) == .primaryHome &&
homes.isEmpty

}

// MARK: HMHomeDelegate Methods

/// Finds the home in the internal structure and reloads the corresponding row.
override func homeDidUpdateName(_ home: HMHome) {
    if let index = homes.firstIndex(of: home) {
        let listIndexPath = IndexPath(row: index, section:
HomeListSection.homes.rawValue)

        let primaryIndexPath = IndexPath(row: index, section:
HomeListSection.primaryHome.rawValue)

        tableView.reloadRows(at: [listIndexPath, primaryIndexPath], with:
.automatic)

```

```

    }

    else {

        // Just reload the data since we don't know the index path.

        tableView.reloadData()

    }

}

}

// MARK: HMHomeManagerDelegate Methods

/// Reloads the 'primary home' section.

func homeManagerDidUpdatePrimaryHome(_ manager: HMHomeManager) {

    didUpdatePrimaryHome()

}

}

}

HomeListConfigurationViewController

import HomeKit

```

A superclass for all trigger creators.

These classes manage the temporary state of the trigger
and unify some of the saving processes.

*/

```
class TriggerCreator {
```

```
// MARK: Properties
```

```
internal var home: HMHome
```

```
internal var trigger: HMTrigger?
```

```
internal var name = ""
```

```
internal let saveTriggerGroup = DispatchGroup()
```

```
internal var errors = [Error]()
```

```
/**
```

Initializes a trigger creator from an existing trigger (if it exists),

and the current home.

- parameter trigger: An HMTrigger or nil, if creation is desired.

- parameter home: The HMHome into which this trigger will go.

*/

required init(trigger: HMTrigger?, home: HMHome) {

self.home = home

self.trigger = trigger

}

/**

Completes one of two actions based on the current status of the trigger object:

1. Updates the existing trigger.
2. Creates a new trigger.

- parameter name: The name to set for the new or updated trigger.

- parameter actionSets: The new list of action sets to set for the trigger

- parameter completion: The closure to call when all configurations have been completed.


```

*/

func saveTriggerWithName(_ name: String, actionSets: [HMAActionSet],
completion: @escaping
(HMTrigger?, [Error]) -> Void) {

    self.name = name

    if trigger != nil {

        // Let the subclass update the trigger.

        updateTrigger()
        updateNameIfNecessary()
        configureWithActionSets(actionSets)
    }
    else {

        self.trigger = newTrigger()

        saveTriggerGroup.enter()

        home.addTrigger(trigger!) { error in

            if let error = error {

                self.errors.append(error)

            }
        }
    }
}

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

else {

    self.configureWithActionSets(actionSets)

}

self.saveTriggerGroup.leave()

}

}

/*
Call the completion block with our event trigger and any accumulated errors
from the saving process.
*/

saveTriggerGroup.notify(queue: DispatchQueue.main) {

    self.cleanUp()

    completion(self.trigger, self.errors)

}

}

```

Updates the trigger's internals.

Action sets and the trigger name need not be configured.

Implemented by subclasses.

*/

internal func updateTrigger() { }

/**

Creates a new trigger to be added to the home.

Action sets and the trigger name need not be configured.

Implemented by subclasses.

- returns: A new, generated HMTrigger.

*/

internal func newTrigger() -> HMTrigger? {

return nil

}

```
/**
```

Cleans up an internal structures after the trigger has been saved.

Implemented by subclasses.

```
*/
```

```
internal func cleanUp() {}
```

```
// MARK: Helper Methods
```

```
/**
```

Syncs the trigger's action sets with the specified array of action sets.

- parameter actionSets: Array of HMAActionSets to match.

```
*/
```

```
private func configureWithActionSets(_ actionSets: [HMAActionSet]) {
```

```
    guard let trigger = trigger else { return }
```

```
/*
```

Save a standard completion handler to use when we either add or remove
an action set.

```
*/
```

```
let defaultCompletion: (Error?) -> Void = { error in
```

```
    // Leave the dispatch group, to notify that we've finished this task.
```

```
    if let error = error {
```

```
        self.errors.append(error)
```

```
    }
```

```
    self.saveTriggerGroup.leave()
```

```
}
```

```
    // First pass, remove the action sets that have been deselected.
```

```
    for actionSet in trigger.actionSets {
```

```
        if actionSets.contains(actionSet) {
```

```
            continue
```

```
        }
```

```
        saveTriggerGroup.enter()
```

```

trigger.removeActionSet(actionSet, completionHandler: defaultCompletion)

}

// Second pass, add the new action sets that were just selected.

for actionSet in actionSets {

    if trigger.actionSets.contains(actionSet) {

        continue

    }

    saveTriggerGroup.enter()

    trigger.addActionSet(actionSet, completionHandler: defaultCompletion)

}

}

/// Updates the trigger's name from the stored name, entering and leaving the
dispatch group if

necessary.

func updateNameIfNecessary() {

    if trigger?.name == self.name {

```



```

        return

    }

    saveTriggerGroup.enter()

    trigger?.updateName(name) { error in

        if let error = error {

            self.errors.append(error)

        }

        self.saveTriggerGroup.leave()

    }

}

}

TriggerCreator

import UIKit

import HomeKit

/// A view controller which facilitates the creation of timer triggers.

class TimerTriggerViewController: TriggerViewController {

    // MARK: Types

```

```
struct Identifiers {
```

```
    static let recurrenceCell = "RecurrenceCell"
```

```
}
```

```
static let RecurrenceTitles = [
```

```
    NSLocalizedString("Every Hour", comment: "Every Hour"),
```

```
    NSLocalizedString("Every Day", comment: "Every Day"),
```

```
    NSLocalizedString("Every Week", comment: "Every Week")
```

```
]
```

```
// MARK: Properties
```

```
@IBOutlet weak var datePicker: UIDatePicker!
```

```
/**
```

```
    Sets the stored fireDate to the new value.
```

```
    HomeKit only accepts dates aligned with minute boundaries.
```

so we use NSDateComponents to only get the appropriate pieces of information from that

date.

Eventually we will end up with a date following this format: "MM/dd/yyyy hh:mm"

*/

```
var timerTrigger: HMTimerTrigger? {  
    return trigger as? HMTimerTrigger  
}
```

```
var timerTriggerCreator: TimerTriggerCreator {  
    return triggerCreator as! TimerTriggerCreator  
}
```

// MARK: View Methods

/// Configures the views and registers for table view cells.

```

override func viewDidLoad() {

    super.viewDidLoad()

    tableView.rowHeight = UITableView.automaticDimension

    tableView.estimatedRowHeight = 44.0

    triggerCreator = TimerTriggerCreator(trigger: trigger, home: home)

    datePicker.date = timerTriggerCreator.fireDate as Date

    tableView.register(UITableViewCell.self, forCellReuseIdentifier:
Identifiers.recurrenceCell)
}

// MARK: IBAction Methods

/// Reset our saved fire date to the date in the picker.

@IBAction func didChangeDate(_ picker: UIDatePicker) {

    timerTriggerCreator.rawFireDate = picker.date

}

// MARK: Table View Methods

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

/**

- returns: The number of rows in the Recurrence section;

defaults to the super implementation for other sections

*/

override func tableView(_ tableView: UITableView, numberOfRowsInSectionSection
section: Int) -> Int {

switch sectionForIndex(section) {

case .recurrence?:

return TimerTriggerViewController.RecurrenceTitles.count

case nil:

fatalError("Unexpected TriggerTableViewSection raw value.")

default:

return super.tableView(tableView, numberOfRowsInSectionSection: section)

}

}

```

```

/**

Generates a recurrence cell.

Defaults to the super implementation for other sections

*/

override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) ->
UITableViewCell {

switch sectionForIndex(indexPath.section) {

case .recurrence?:

return self.tableView(tableView, recurrenceCellForRowAtIndexPath:
indexPath)

case nil:

fatalError("Unexpected TriggerTableViewSection raw value.")

default:

return super.tableView(tableView, cellForRowAt: indexPath)

```



```

    }

}

/// Creates a cell that represents a recurrence type.

func tableView(_ tableView: UITableView, recurrenceCellForRowAtIndexPath
indexPath:
IndexPath) -> UITableViewCell {

    let cell = tableView.dequeueReusableCell(withIdentifier:
Identifiers.recurrenceCell, for:
indexPath)

    let title = TimerTriggerViewController.RecurrenceTitles[indexPath.row]
    cell.textLabel?.text = title

    // The current preferred recurrence style should have a check mark.

    if indexPath.row == timerTriggerCreator.selectedRecurrenceIndex {

        cell.accessoryType = .checkmark

    }

    else {

```

```

        cell.accessoryType = .none

    }

    return cell

}

/**
Tell the tableView to automatically size the custom rows, while using the
superclass's
static sizing for the static cells.
*/
override func tableView(_ tableView: UITableView, heightForRowAt indexPath:
IndexPath) ->
CGFloat {

    switch sectionForIndex(indexPath.section) {

        case .recurrence?:

            return UITableView.automaticDimension

        case nil:

```

```

fatalError("Unexpected TriggerTableViewSection raw value.")

default:

return super.tableView(tableView, heightForRowAt: indexPath)

}

}

/**
Handles recurrence cell selection.
Defaults to the super implementation for other sections
*/

override func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {

tableView.deselectRow(at: indexPath, animated: true)

switch sectionForIndex(indexPath.section) {

case .recurrence?:

self.tableView(tableView, didSelectRecurrenceComponentAtIndex:
indexPath)

```

case nil:

```
fatalError("Unexpected TriggerTableViewSection raw value.")
```

default:

```
super.tableView(tableView, didSelectRowAt: indexPath)
```

```
}
```

```
}
```

```
/**
```

Handles selection of a recurrence cell.

If the newly selected recurrence component is the previously selected

recurrence component, reset the current selected component to NSNotFound

and deselect that row.

```
*/
```

```
func tableView(_ tableView: UITableView, didSelectRecurrenceComponentAtIndex:
```

```

indexPath: IndexPath) {

    if indexPath.row == timerTriggerCreator.selectedRecurrenceIndex {

        timerTriggerCreator.selectedRecurrenceIndex = nil

    }

    else {

        timerTriggerCreator.selectedRecurrenceIndex = indexPath.row

    }

    tableView.reloadSections(IndexSet(integer: indexPath.section),
    with:
    .automatic)

}

/**
- parameter index: The section index.

- returns: The TriggerTableViewSection for the given index.

*/

override func sectionForIndex(_ index: Int) -> TriggerTableViewSection? {

    switch index {

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

case 0:

return .name

case 1:

return .enabled

case 2:

return .dateAndTime

case 3:

return .recurrence

case 4:

return .actionSets

default:

return nil

}

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้


```

    }

}

TimerTriggeViewController

import HomeKit

/**
A TriggerCreator subclass which allows for the creation
of timer triggers.
*/
class TimerTriggerCreator: TriggerCreator {

    static let RecurrenceComponents: [Calendar.Component] = [

        .hour,

        .day,

        .weekOfYear

    ]

}

```

// MARK: Properties

```

var timerTrigger: HMTimerTrigger? {

    return trigger as? HMTimerTrigger

}

var selectedRecurrenceIndex: Int? = nil

var rawFireDate = Date()
var fireDate: Date {

    let flags: Set<Calendar.Component> = [.year, .weekday, .month, .day, .hour,
.minute]

    let dateComponents = Calendar.current.dateComponents(flags, from:
self.rawFireDate)

    let probableDate = Calendar.current.date(from: dateComponents)

    return probableDate ?? rawFireDate

}

// MARK: Trigger Creator Methods

```

```

/// Configures raw fire date and selected recurrence index.

required init(trigger: HMTrigger?, home: HMHome) {

    super.init(trigger: trigger, home: home)

    if let timerTrigger = timerTrigger {

        rawFireDate = timerTrigger.fireDate

        selectedRecurrenceIndex =
recurrenceIndexFromDateComponents(timerTrigger.recurrence)
    }
}

/// - returns: A new HMTimerTrigger with the stored configurations.

override func newTrigger() -> HMTrigger? {

    return HMTimerTrigger(name: name, fireDate: fireDate, timeZone:

Calendar.current.timeZone,          recurrence:          recurrenceComponents,
recurrenceCalendar: nil)

}

```

```
/// Updates the fire date and recurrence of the trigger.
```

```
override func updateTrigger() {
```

```
    updateFireDateIfNecessary()
```

```
    updateRecurrenceIfNecessary()
```

```
}
```

```
// MARK: Helper Methods
```

```
/**
```

Creates an NSDateComponent for the selected recurrence type.

- returns: An NSDateComponent where either weekOfYear,
hour, or day is set to 1.

```
*/
```

```
var recurrenceComponents:DateComponents? {
```

```
    guard let selectedRecurrenceIndex = selectedRecurrenceIndex else {
```

```
        return nil
```

```
    }
```

```

var recurrenceComponents = DateComponents()

let unit =

TimerTriggerCreator.RecurrenceComponents[selectedRecurrenceIndex]

switch unit {

case .weekOfYear:

recurrenceComponents.weekOfYear = 1

case .hour:

recurrenceComponents.hour = 1

case .day:

recurrenceComponents.day = 1

default:

break

}

return recurrenceComponents

}

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และส่งต่อหรือแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

/**

Maps the possible calendar units associated with recurrence titles, so we can properly

set our recurrenceUnit when an index is selected.

- parameter components: An optional NSDateComponents to query.

- returns: An index for the date components.

*/

```
func recurrenceIndexFromDateComponents(_ components: DateComponents?) -  
> Int? {
```

```
    guard let components = components else { return nil }
```

```
    var unit: Calendar.Component?
```

```
    if components.day == 1 {
```

```
        unit = .day
```

```
    }
```

```
    else if components.weekOfYear == 1 {
```



```

        unit = .weekOfYear
    }

    else if components.hour == 1 {

        unit = .hour

    }

    if let unit = unit {

        return TimerTriggerCreator.RecurrenceComponents.firstIndex(of: unit)

    }

    return nil

}

/**

```

Updates the trigger's fire date, entering and leaving the dispatch group if necessary.

If the trigger's fire date is already equal to the passed-in fire date, this method does nothing.

- parameter fireDate: The trigger's new fire date.

```
*/
```

```
private func updateFireDateIfNecessary() {
```

```
    if timerTrigger?.fireDate == fireDate {
```

```
        return
```

```
    }
```

```
    saveTriggerGroup.enter()
```

```
    timerTrigger?.updateFireDate(fireDate) { error in
```

```
        if let error = error {
```

```
            self.errors.append(error)
```

```
        }
```

```
        self.saveTriggerGroup.leave()
```

```
    }
```

```
}
```

```
/**
```

Updates the trigger's recurrence components, entering and leaving the dispatch group if

necessary.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 292 จึงอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

If the trigger's components are already equal to the passed-in components, this method does

nothing.

- parameter recurrenceComponents: The trigger's new recurrence components.

*/

```
private func updateRecurrenceIfNecessary() {  
    if recurrenceComponents == timerTrigger?.recurrence {  
        return  
    }  
    saveTriggerGroup.enter()  
    timerTrigger?.updateRecurrence(recurrenceComponents) { error in  
        if let error = error {  
            self.errors.append(error)  
        }  
        self.saveTriggerGroup.leave()  
    }  
}
```

```
}
```

```
TimerTriggerCreator
```

```
import UIKit
```

```
import HomeKit
```

```
/// Represents all possible sections in a TriggerViewController subclass.
```

```
enum TriggerTableViewSection: Int {
```

```
    // All triggers have these sections.
```

```
    case name, enabled, actionSets
```

```
    // Timer triggers only.
```

```
    case dateAndTime, recurrence
```

```
    // Location and Characteristic triggers only.
```

```
    case conditions
```

```
    // Location triggers only.
```

```
    case location, region
```

```

// Characteristic triggers only.

case characteristics

}

/**
A superclass for all trigger view controllers.

It manages the name, enabled state, and action set components of the view,
as these are shared components.
*/
class TriggerViewController: HMCatalogViewController {

// MARK: Types

struct Identifiers {

static let actionSetCell = "ActionSetCell"

}

```

```
// MARK: Properties
```

```
@IBOutlet weak var saveButton: UIBarButtonItem!
```

```
@IBOutlet weak var nameField: UITextField!
```

```
@IBOutlet weak var enabledSwitch: UISwitch!
```

```
var trigger: HMTrigger?
```

```
var triggerCreator: TriggerCreator?
```

```
/// An internal array of all action sets in the home.
```

```
var actionSets: [HMActionSet]!
```

```
/**
```

An array of all action sets that the user has selected.

This will be used to save the trigger when it is finalized.

```
*/
```

```
lazy var selectedActionSets = [HMActionSet]()
```



```
// MARK: View Methods
```

```
/// Resets internal data, sets initial UI, and configures the table view.
```

```
override func viewDidLoad() {
```

```
    super.viewDidLoad()
```

```
    let filteredActionSets = home.actionSets.filter { actionSet in
```

```
        return !actionSet.actions.isEmpty
```

```
    }
```

```
    actionSets = filteredActionSets.sortByTypeAndLocalizedString()
```

```
    tableView.rowHeight = UITableView.automaticDimension
```

```
    tableView.estimatedRowHeight = 44.0
```

```
    /*
```

If we have a trigger, set the saved properties to the current properties

of the passed-in trigger.

```
    */
```

```
    if let trigger = trigger {
```

```

selectedActionSets = trigger.actionSets

nameField.text = trigger.name

enabledSwitch.isOn = trigger.isEnabled

}

```

```

enableSaveButtonIfApplicable()

```

```

tableView.register(UITableViewCell.self, forCellReuseIdentifier:
Identifiers.actionSetCell)
}

```

```

// MARK: IBAction Methods

```

```

/**

```

Any time the name field changed, reevaluate whether or not

to enable the save button.

```

*/

```

```

@IBAction func nameFieldDidChange(_ sender: UITextField) {

```

```

enableSaveButtonIfApplicable()

}

/// Saves the trigger and dismisses this view controller.

@IBAction func saveAndDismiss() {

    saveButton.isEnabled = false

    triggerCreator?.saveTriggerWithName(trimmedName,
selectedActionSets) {
trigger, errors in
        self.trigger = trigger
        self.saveButton.isEnabled = true

        if !errors.isEmpty {

            self.displayErrors(errors)

            return

        }

        self.enableTrigger(self.trigger!) {

```

```

        self.dismiss()

    }

}

}

@IBAction func dismiss() {

    self.dismiss(animated: true, completion: nil)

}

// MARK: Subclass Methods

/**

Generates the section for the index.

```

This allows for the subclasses to lay out their content in different sections

while still maintaining common code in the TriggerViewController.

- parameter index: The index of the section

- returns: The TriggerTableViewSection for the provided index.

*/

```
func sectionForIndex(_ index: Int) -> TriggerTableViewSection? {
```

```
    return nil
```

```
}
```

```
// MARK: Helper Methods
```

```
/// Enable the trigger if necessary.
```

```
func enableTrigger(_ trigger: HMTrigger, completion: @escaping () -> Void) {
```

```
    if trigger.isEnabled == enabledSwitch.isOn {
```

```
        completion()
```

```
    return
```

```
}
```

```
trigger.enable(enabledSwitch.isOn) { error in
```

```
    if let error = error {
```

```

        self.displayError(error)

    }

    else {

        completion()

    }

}

}

}

```

```
/**
```

Enables the save button if:

1. The name field is not empty, and
2. There will be at least one action set in the trigger after saving.

```
*/
```

```
private func enableSaveButtonIfApplicable() {
```

```
    saveButton.isEnabled = !trimmedName.isEmpty &&
```

```
    (!selectedActionSets.isEmpty || trigger?.actionSets.count ?? 0 > 0)
```

```
}
```


/// - returns: The name from the nameField, stripping newline and whitespace characters.

```
var trimmedName: String {
```

```
    return nameField.text!.trimmingCharacters(in: CharacterSet.whitespacesAndNewlines)
```

```
}
```

// MARK: Table View Methods

/// Creates a cell that represents either a selected or unselected action set cell.

```
private func tableView(_ tableView: UITableView,
    actionSetCellForRowAtIndexPath indexPath:
```

```
IndexPath) -> UITableViewCell {
```

```
    let cell = tableView.dequeueReusableCell(withIdentifier:
    Identifiers.actionSetCell, for:
```

```
indexPath)
```

```
    let actionSet = actionSets[indexPath.row]
```

```

if selectedActionSets.contains(actionSet) {

    cell.accessoryType = .checkmark

}

else {

    cell.accessoryType = .none

}

cell.textLabel?.text = actionSet.name

return cell

}

/// Only handles the ActionSets case, defaults to super.

override func tableView(_ tableView: UITableView, numberOfRowsInSectionSection
section: Int) -> Int {

    if sectionForIndex(section) == .actionSets {

        return actionSets?.count ?? 0
    }

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

    }

    return super.tableView(tableView, numberOfRowsInSection: section)

}

/// Only handles the ActionSets case, defaults to super.

override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) ->
UITableViewCell {

    if sectionForIndex(indexPath.section) == .actionSets {

        return self.tableView(tableView, actionSetCellForRowAtIndexPath:
indexPath)

    }

    return super.tableView(tableView, cellForRowAt: indexPath)

}

/**

```

This is necessary for mixing static and dynamic table view cells.

We return a fake index path because otherwise the superclass's implementation (which does

not

know about the extra cells we're adding) will cause an error.

- returns: The superclass's `indentationLevel` for the first row in the provided section,

instead of the provided row.

*/

override func tableView(_ tableView: UITableView, indentationLevelForRowAt indexPath:

IndexPath) -> Int {

let newIndexPath = IndexPath(row: 0, section: indexPath.section)

return super.tableView(tableView, indentationLevelForRowAt: newIndexPath)

}

/**

Tell the tableView to automatically size the custom rows, while using the superclass's

static sizing for the static cells.

*/

override func tableView(_ tableView: UITableView, heightForRowAt indexPath: IndexPath) ->

CGFloat {

switch sectionForIndex(indexPath.section) {

case .name?, .enabled?:

return super.tableView(tableView, heightForRowAt: indexPath)

case nil:

fatalError("Unexpected TriggerTableViewSection raw value.")

default:

return UITableView.automaticDimension

}

}

```

/// Handles row selection for action sets, defaults to super implementation.

override func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {

    tableView.deselectRow(at: indexPath, animated: true)

    if sectionForIndex(indexPath.section) == .actionSets {

        self.tableView(tableView, didSelectActionSetAtIndex: indexPath)
    }
}

/**
Manages footer titles for higher-level sections. Superclasses should fall back
on this implementation after attempting to handle any special trigger sections.

*/

override func tableView(_ tableView: UITableView, titleForFooterInSection
section: Int) ->

String? {

    switch sectionForIndex(section) {

```



```
case .actionSets?:
```

```
    return NSLocalizedString("When this trigger is activated, it will set these  
scenes. You can
```

```
only select scenes which have at least one action.", comment: "Scene Trigger  
Description")
```

```
case .enabled?:
```

```
    return NSLocalizedString("This trigger will only activate if it is enabled.  
You can disable
```

```
triggers to temporarily stop them from running.", comment: "Trigger Enabled  
Description")
```

```
case nil:
```

```
    fatalError("Unexpected TriggerTableViewSection raw value.")
```

```
default:
```

```
    return super.tableView(tableView, titleForFooterInSection: section)
```

```
}
```

```
}
```

```
/**
```

Handle selection of an action set cell. If the action set is already part of the selected action

sets,

then remove it from the selected list. Otherwise, add it to the selected list.

```
*/
```

```
func tableView(_ tableView: UITableView, didSelectActionSetAtIndex  
indexPath: IndexPath)
```

```
{
```

```
let actionSet = actionSets[indexPath.row]
```

```
if let index = selectedActionSets.firstIndex(of: actionSet) {
```

```
selectedActionSets.remove(at: index)
```

```
}
```

```
else {
```

```
selectedActionSets.append(actionSet)
```

```
}
```

```

enableSaveButtonIfApplicable()

tableView.reloadRows(at: [indexPath], with: .automatic)

}

// MARK: HMHomeDelegate Methods

/**
If our trigger has been removed from the home,
dismiss the view controller.
*/
func home(_ home: HMHome, didRemoveTrigger trigger: HMTrigger) {

    if self.trigger == trigger{

        self.dismiss(animated: true, completion: nil)

    }

}

/// If our trigger has been updated, reload our data.

func home(_ home: HMHome, didUpdateTrigger trigger: HMTrigger) {

```

```

        if self.trigger == trigger{

            tableView.reloadData()

        }

    }

}

```

TriggerViewController

```
import UIKit
```

```
import HomeKit
```

```
/**
```

A superclass for all event-based view controllers.

It handles the process of creating and managing trigger conditions.

```
*/
```

```
class EventTriggerViewController: TriggerViewController {
```

```
    // MARK: Types
```

```

struct Identifiers {

    static let addCell = "AddCell"

    static let conditionCell = "ConditionCell"

    static let showTimeConditionSegue = "Show Time Condition"

}

// MARK: Properties

private var eventTriggerCreator: EventTriggerCreator {

    return triggerCreator as! EventTriggerCreator

}

// MARK: View Methods

/// Registers table view for cells.

override func viewDidLoad() {

    super.viewDidLoad()

```

```
tableView.register(UITableViewCell.self,  
forCellReuseIdentifier:Identifiers.addCell)
```

```
tableView.register(ConditionCell.self,  
forCellReuseIdentifier:Identifiers.conditionCell)
```

```
}
```

```
/// Hands off the trigger creator to the condition view controllers.
```

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
```

```
switch segue.intendedDestinationViewController {
```

```
case let timeVC as TimeConditionViewController:
```

```
timeVC.triggerCreator = eventTriggerCreator
```

```
case let characteristicEventVC as CharacteristicSelectionViewController:
```

```
let characteristicTriggerCreator = triggerCreator as! EventTriggerCreator
```

```
characteristicEventVC.triggerCreator = characteristicTriggerCreator
```

```
default:
```

```
break
```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้


```

    }

}

// MARK: Table View Methods

/**
 - returns: In the conditions section: the number of conditions, plus one
           for the add row. Defaults to the super implementation.
 */
override func tableView(_ tableView: UITableView, numberOfRowsInSectionSection
section: Int) -> Int {

    switch sectionForIndex(section) {

        case .conditions?:

            // Add row.

            return eventTriggerCreator.conditions.count + 1

        case nil:

            fatalError("Unexpected TriggerTableViewSection raw value.")

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

default:

```
return super.tableView(tableView, numberOfRowsInSection: section)
```

```
}
```

```
}
```

```
/**
```

Launchs "Add Condition" if the 'add index path' is selected.

Defaults to the super implementation.

```
*/
```

```
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
```

```
tableView.deselectRow(at: indexPath, animated: true)
```

```
switch sectionForIndex(indexPath.section) {
```

```
case .conditions?:
```

```
if indexPathIsAdd(indexPath) {
```

```
addCondition()
```

```
}
```

case nil:

```
fatalError("Unexpected TriggerTableViewSection raw value.")
```

default:

```
super.tableView(tableView, didSelectRowAt: indexPath)
```

```
}
```

```
}
```

```
/**
```

Switches to select the correct type of cell for the section.

Defaults to the super implementation.

```
*/
```

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
```

```
UITableViewCell {
```

```
    if indexPathIsAdd(indexPath) {
```

```
        return self.tableView(tableView, addCellForRowAtIndexPath: indexPath)
```

```

    }

    switch sectionForIndex(indexPath.section) {

        case .conditions?:

            return self.tableView(tableView, conditionCellForRowAtIndexPath:
indexPath)

        case nil:

            fatalError("Unexpected TriggerTableViewSection raw value.")

        default:

            return super.tableView(tableView, cellForRowAt: indexPath)

    }

}

/**

```

The conditions can be removed, the 'add index path' cannot.

For all others, default to super implementation.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

*/

override func tableView(_ tableView: UITableView, canEditRowAt indexPath:
IndexPath) -> Bool
{

    if indexPathIsAdd(indexPath) {

        return false

    }

    switch sectionForIndex(indexPath.section) {
    case .conditions?:
        return true

    case nil:
        fatalError("Unexpected TriggerTableViewSection raw value.")

    default:

        return false

    }
}

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

}

/// Remove the selected condition from the trigger creator.

override func tableView(_ tableView: UITableView, commit editingStyle:
UITableViewCellEditingStyle, forRowAt indexPath: IndexPath) {

    if editingStyle == .delete {

        let predicate = eventTriggerCreator.conditions[indexPath.row]

        eventTriggerCreator.removeCondition(predicate)

        tableView.deleteRows(at: [indexPath], with: .automatic)

    }

}

/// - returns: An 'add cell' with 'Add Condition' text.

func tableView(_ tableView: UITableView, addCellForRowAtIndexPath
indexPath: IndexPath) ->

UITableViewCell {

    let cell = tableView.dequeueReusableCell(withIdentifier: Identifiers.addCell,
for: indexPath)

```



```

let cellText: String

switch sectionForIndex(indexPath.section) {

    case .conditions?:

        cellText = NSLocalizedString("Add Condition...", comment: "Add
Condition")

    case nil:

        fatalError("Unexpected TriggerTableViewSection raw value.")

    default:

        cellText = NSLocalizedString("Add...", comment: "Add")
}

cell.textLabel?.text = cellText

cell.textLabel?.textColor = .editableBlue

return cell
}

```

/// - returns: A localized description of a trigger. Falls back to super implementation.

override func tableView(_ tableView: UITableView, titleForFooterInSection section: Int) ->

String? {

switch sectionForIndex(section) {

case .conditions?:

return NSLocalizedString("When a trigger is activated by an event, it checks these

conditions. If all of them are true, it will set its scenes.", comment: "Trigger Conditions

Description")

case nil:

fatalError("Unexpected TriggerTableViewSection raw value.")

default:

return super.tableView(tableView, titleForFooterInSection: section)

```

    }

}

// MARK: Helper Methods

/// - returns: A 'condition cell', which displays information about the condition.

private func tableView(_ tableView: UITableView,
conditionCellForRowAtIndexPath indexPath:
IndexPath) -> UITableViewCell {

    let cell = tableView.dequeueReusableCell(withIdentifier:
Identifiers.conditionCell) as!
ConditionCell

    let condition = eventTriggerCreator.conditions[indexPath.row]

    switch condition.homeKitConditionType {

    case .characteristic(let characteristic, let value):

        cell.setCharacteristic(characteristic, targetValue: value)

```

```
case .exactTime(let order, let dateComponents):
```

```
    cell.setOrder(order, dateComponents: dateComponents)
```

```
case .sunTime(let order, let sunState):
```

```
    cell.setOrder(order, sunState: sunState)
```

```
case .unknown:
```

```
    cell.setUnknown()
```

```
}
```

```
return cell
```

```
}
```

```
/// Presents an alert controller to choose the type of trigger.
```

```
private func addCondition() {
```

```
    let title = NSLocalizedString("Add Condition", comment: "Add Condition")
```

```

let alertController = UIAlertController(title: title, message: nil, preferredStyle:
.actionSheet)

// Time Condition.

let timeAction = UIAlertAction(title: NSLocalizedString("Time", comment:
"Time"), style:
.default) { _ in

    self.performSegue(withIdentifier: Identifiers.showTimeConditionSegue,
sender: self)
}

alertController.addAction(timeAction)

// Characteristic trigger.

let eventActionTitle = NSLocalizedString("Characteristic", comment:
"Characteristic")

let eventAction = UIAlertAction(title: eventActionTitle, style: .default, handler:
{ _ in

    if let triggerCreator = self.triggerCreator as? CharacteristicTriggerCreator {

        triggerCreator.mode = .condition

```

```
}  
  
self.performSegue(withIdentifier: "Select Characteristic", sender: self)  
  
})
```

```
alertController.addAction(eventAction)  
  
// Cancel.  
let cancelAction = UIAlertAction(title: NSLocalizedString("Cancel", comment:  
"Cancel"), style:  
.cancel, handler: nil)  
alertController.addAction(cancelAction)  
  
// Present alert.  
  
if let presenter = alertController.popoverPresentationController {  
  
    presenter.sourceView = self.view  
  
    presenter.sourceRect = self.view.bounds  
  
}
```



```

        present(alertController, animated: true, completion: nil)

    }

    /// - returns: true if the index path is the 'add row'; false otherwise.

    func indexPathIsAdd(_ indexPath: IndexPath) -> Bool {

        switch sectionForIndex(indexPath.section) {

        case .conditions?:

            return indexPath.row == eventTriggerCreator.conditions.count

        case nil:

            fatalError("Unexpected TriggerTableViewSection raw value.")

        default:

            return false

        }

    }

}

```

EventTriggerViewController

```

import UIKit

import HomeKit

/// Represents modes for a CharacteristicTriggerCreator.

enum CharacteristicTriggerCreatorMode: Int {

    case event, condition

}

/**
An EventTriggerCreator subclass which allows for the creation
of characteristic triggers.

*/

class CharacteristicTriggerCreator: EventTriggerCreator {

    // MARK: Properties

    var eventTrigger: HMEventTrigger? {

        return self.trigger as? HMEventTrigger

    }

```

/**

This object will be a characteristic cell delegate and will therefore be receiving updates when UI elements change value. However, this object can construct both characteristic events and characteristic triggers.

Setting the mode determines how this trigger creator will handle cell delegate callbacks.

*/

```
var mode: CharacteristicTriggerCreatorMode = .event
```

/**

Contains the new pending mapping of HMCharacteristics to their trigger (NSCopying) values.

When `saveTriggerWithName(name:completion:)` is called, all of these mappings will be

converted

into `HMCharacteristicEvents` and added to the `HMEventTrigger`.

*/

```

private let targetValueMap = NSMapTable<HMCharacteristic,
CellValueType>.strongToStrongObjects()

/// HMCharacteristicEvents that should be removed if
saveTriggerWithName(name:completion:) is called.

private          var          removalCharacteristicEvents          =
[HMCharacteristicEvent<NSCopying>]()

// MARK: Trigger Creator Methods

/// Syncs the stored event trigger using internal values.

override func updateTrigger() {

    guard let eventTrigger = eventTrigger else { return }

    matchEventsFromTriggerIfNecessary()

    removePendingEventsFromTrigger()

    for (characteristic, triggerValue) in pairsFromMapTable(targetValueMap) {

        let newEvent = HMCharacteristicEvent(characteristic: characteristic,
triggerValue:

```

```
triggerValue)
```

```
self.saveTriggerGroup.enter()
```

```
eventTrigger.addEvent(newEvent) { error in
```

```
if let error = error {
```

```
self.errors.append(error)
```

```
}
```

```
self.saveTriggerGroup.leave()
```

```
}
```

```
}
```

```
savePredicate()
```

```
}
```

```
/**
```

- returns: A new HMEventTrigger with the pending

characteristic events and constructed predicate.

```
*/
```

```
override func newTrigger() -> HMTrigger? {
```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```
return HMEventTrigger(name: name, events: pendingCharacteristicEvents,  
predicate:
```

```
newPredicate())
```

```
}
```

```
/**
```

Remove all objects from the map so they don't show up
in the events computed array.

```
*/
```

```
override func cleanUp() {
```

```
targetValueMap.removeAllObjects()
```

```
}
```

```
/**
```

Removes an event from the map table if it's a new event and
queues it for removal if it already existed in the event trigger.

- parameter event: HMCharacteristicEvent to be removed.


```

*/

func removeEvent(_ event: HMCharacteristicEvent<CellValueType>) {

    if targetValueMap.object(forKey: event.characteristic) != nil {

        // Remove the characteristic from the target value map.

        targetValueMap.removeObject(forKey: event.characteristic)

    }

    if let characteristicEvents = eventTrigger?.characteristicEvents ,
characteristicEvents.contains(event) {

        // If the given event is in the event array, queue it for removal.

        removalCharacteristicEvents.append(event)

    }

}

// MARK: Helper Methods

/**

```

Any characteristic events in the map table that have not yet been

added to the trigger.

*/

```
var pendingCharacteristicEvents: [HMCharacteristicEvent<CellValueType>] {
```

```
    let pairs = pairsFromMapTable(targetValueMap)
```

```
    return pairs.map { (characteristic, triggerValue) ->
        HMCharacteristicEvent<CellValueType> in
```

```
        return HMCharacteristicEvent(characteristic: characteristic, triggerValue:
            triggerValue)
```

```
    }
```

```
}
```

```
/**
```

Loops through the characteristic events in the trigger.

If any characteristics in our map table are also in the event,

replace the value with the one we have stored and remove that entry from

our map table.

*/

```
private func matchEventsFromTriggerIfNecessary() {
```

```

guard let eventTrigger = eventTrigger else { return }

for event in eventTrigger.characteristicEvents {

    // Find events who's characteristic is in our map table.

    if let triggerValue = targetValueMap.object(forKey: event.characteristic) {

        self.saveTriggerGroup.enter()

        event.updateTriggerValue(triggerValue) { error in

            if let error = error {

                self.errors.append(error)

            }

            self.saveTriggerGroup.leave()

        }

    }

}

}

}

/**

```

Removes all HMCharacteristicEvents from the removalCharacteristicEvents

array and stores any errors that accumulate.

*/

```
private func removePendingEventsFromTrigger() {  
  
    guard let eventTrigger = eventTrigger else { return }  
  
    for event in removalCharacteristicEvents {  
  
        saveTriggerGroup.enter()  
  
        eventTrigger.removeEvent(event) { error in  
  
            if let error = error {  
  
                self.errors.append(error)  
            }  
  
            self.saveTriggerGroup.leave()  
        }  
  
    }  
  
    removalCharacteristicEvents.removeAll()  
  
}
```

/// All HMCharacteristics in the targetValueMap.

```

private var allCharacteristics: [HMCharacteristic] {

    var characteristics = Set<HMCharacteristic>()

    for characteristic in targetValueMap.keyEnumerator().allObjects as!
[HMCharacteristic] {

        characteristics.insert(characteristic)

    }

    return Array(characteristics)
}

/**
Saves a characteristic and value into the pending map
of characteristic events.

- parameter value: The value of the characteristic.

- parameter characteristic: The HMCharacteristic that has been updated.

*/

private func updateEventValue(_ value: NSCopying, forCharacteristic
characteristic:

```

```
HMCharacteristic) {
```

```
    for (index, event) in removalCharacteristicEvents.enumerated() {
```

```
        if event.characteristic == characteristic {
```

```
            /*
```

We have this event pending for deletion,

but we are going to want to update it.

remove it from the removal array.

```
        */
```

```
        removalCharacteristicEvents.remove(at: index)
```

```
        break
```

```
    }
```

```
}
```

```
targetValueMap.setObject(value, forKey: characteristic)
```

```
}
```

```
/**
```

The current, sorted collection of HMCharacteristicEvents accumulated by

filtering out the events pending removal from the original trigger events and

then adding new pending events.

*/

```
var events: [HMCharacteristicEvent<CellValueType>] {
```

```
    let characteristicEvents = eventTrigger?.characteristicEvents ?? []
```

```
    let originalEvents = characteristicEvents.filter {
```

```
        return !removalCharacteristicEvents.contains($0)
```

```
    }
```

```
    let allEvents = originalEvents + pendingCharacteristicEvents
```

```
    return allEvents.sorted { (event1: HMCharacteristicEvent, event2: HMCharacteristicEvent) in
```

```
        let type1 = event1.characteristic.localizedCharacteristicType
```

```
        let type2 = event2.characteristic.localizedCharacteristicType
```

```
        return type1.localizedCompare(type2) == .orderedAscending
```

```
    }
```

```
}
```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และส่งต่อ339 ไปถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

// MARK: CharacteristicCellDelegate Methods

/**

If the mode is event, update the event value.

Otherwise, default to super implementation

*/

override func characteristicCell(_ cell: CharacteristicCell, didUpdateValue value:
CellValueType,
forCharacteristic characteristic: HMCharacteristic, immediate: Bool) {
    switch mode {
    case .event:
        updateEventValue(value, forCharacteristic: characteristic)

    default:
        super.characteristicCell(cell, didUpdateValue: value, forCharacteristic:
characteristic,
immediate: immediate)

```

```

    }

}

/**
    Tries to find the characteristic in either the event map or the
    condition map (based on the current mode). Then calls read value.

    When the value comes back, we check the selected map for the value

*/
override fun characteristicCell(_ cell: CharacteristicCell,
readInitialValueForCharacteristic
characteristic: HMCharacteristic, completion: @escaping (CellValueType?, Error?)
-> Void) {
    if mode == .condition {
        // This is a condition, fall back to the EventTriggerCreator read.

        super.characteristicCell(cell, readInitialValueForCharacteristic:
characteristic, completion:
completion)

        return
    }
}

```

```

if let value = targetValueMap.object(forKey: characteristic) {

    completion(value, nil)

    return

}

characteristic.readValue { error in

    /*
    The user may have updated the cell value while the
    read was happening. We check the map one more time.
    */

    if let value = self.targetValueMap.object(forKey: characteristic) {

        completion(value, nil)

    }

    else {

        completion(characteristic.value as? CellValueType, error)

    }

}

```

```

    }

}

CharacteristicTriggerCreator

import UIKit

import HomeKit

/// Represents modes for a CharacteristicTriggerCreator.
enum CharacteristicTriggerCreatorMode: Int {

    case event, condition

}

/**

    An EventTriggerCreator subclass which allows for the creation

    of characteristic triggers.

*/

class CharacteristicTriggerCreator: EventTriggerCreator {

    // MARK: Properties

```

```
var eventTrigger: HMEventTrigger? {  
  
    return self.trigger as? HMEventTrigger  
  
}
```

```
/**
```

This object will be a characteristic cell delegate and will therefore be receiving updates when UI elements change value. However, this object can construct both characteristic events and characteristic triggers. Setting the mode determines how this trigger creator will handle cell delegate callbacks.

```
*/
```

```
var mode: CharacteristicTriggerCreatorMode = .event
```

```
/**
```

Contains the new pending mapping of HMCharacteristics to their trigger (NSCopying)

values.

When `saveTriggerWithName(name:completion:)` is called, all of these mappings will

be converted

into `HMCharacteristicEvents` and added to the `HMEventTrigger`.

*/

```
private let targetValueMap = NSMapTable<HMCharacteristic,  
CellValueType>.strongToStrongObjects()
```

```
/// HMCharacteristicEvents that should be removed if  
saveTriggerWithName(name:completion:) is called.
```

```
private var removalCharacteristicEvents  
[HMCharacteristicEvent<NSCopying>]()
```

=

```
// MARK: Trigger Creator Methods
```

```
/// Syncs the stored event trigger using internal values.
```

```
override func updateTrigger() {
```

```
    guard let eventTrigger = eventTrigger else { return }
```

```

matchEventsFromTriggerIfNecessary()

removePendingEventsFromTrigger()

for (characteristic, triggerValue) in pairsFromMapTable(targetValueMap) {

    let newEvent = HMCharacteristicEvent(characteristic: characteristic,
triggerValue:
triggerValue)

    self.saveTriggerGroup.enter()
    eventTrigger.addEvent(newEvent) { error in
        if let error = error {
            self.errors.append(error)
        }
        self.saveTriggerGroup.leave()
    }
}

savePredicate()

}

/**

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

- returns: A new HMEventTrigger with the pending

characteristic events and constructed predicate.

*/

override func newTrigger() -> HMTrigger? {

return HMEventTrigger(name: name, events: pendingCharacteristicEvents,
predicate:

newPredicate())

}

/**

Remove all objects from the map so they don't show up

in the events computed array.

*/

override func cleanUp() {

targetValueMap.removeAllObjects()

}

/**

Removes an event from the map table if it's a new event and
queues it for removal if it already existed in the event trigger.

- parameter event: HMCharacteristicEvent to be removed.

*/

```
func removeEvent(_ event: HMCharacteristicEvent<CellValueType>) {  
  
    if targetValueMap.object(forKey: event.characteristic) != nil {  
  
        // Remove the characteristic from the target value map.  
        targetValueMap.removeObject(forKey: event.characteristic)  
    }  
  
    if let characteristicEvents = eventTrigger?.characteristicEvents ,  
characteristicEvents.contains(event) {  
  
        // If the given event is in the event array, queue it for removal.  
  
        removalCharacteristicEvents.append(event)  
  
    }  
  
}
```

```
// MARK: Helper Methods
```

```
/**
```

Any characteristic events in the map table that have not yet been

added to the trigger.

```
*/
```

```
var pendingCharacteristicEvents: [HMCharacteristicEvent<CellValueType>] {
```

```
    let pairs = pairsFromMapTable(targetValueMap)
```

```
    return pairs.map { (characteristic, triggerValue) ->
```

```
        HMCharacteristicEvent<CellValueType> in
```

```
            return HMCharacteristicEvent(characteristic: characteristic, triggerValue:
```

```
            triggerValue)
```

```
        }
```

```
    }
```

```
/**
```

Loops through the characteristic events in the trigger.

If any characteristics in our map table are also in the event,

replace the value with the one we have stored and remove that entry from
our map table.

*/

```
private func matchEventsFromTriggerIfNecessary() {  
  
    guard let eventTrigger = eventTrigger else { return }  
  
    for event in eventTrigger.characteristicEvents {  
  
        // Find events who's characteristic is in our map table.  
  
        if let triggerValue = targetValueMap.object(forKey: event.characteristic) {  
            self.saveTriggerGroup.enter()  
            event.updateTriggerValue(triggerValue) { error in  
                if let error = error {  
                    self.errors.append(error)  
                }  
  
                self.saveTriggerGroup.leave()  
  
            }  
  
        }  
  
    }  
  
}
```



```
/**
```

Removes all HMCharacteristicEvents from the removalCharacteristicEvents array and stores any errors that accumulate.

```
*/
```

```
private func removePendingEventsFromTrigger() {
```

```
    guard let eventTrigger = eventTrigger else { return }
```

```
    for event in removalCharacteristicEvents {
```

```
        saveTriggerGroup.enter()
```

```
        eventTrigger.removeEvent(event) { error in
```

```
            if let error = error {
```

```
                self.errors.append(error)
```

```
            }
```

```
        self.saveTriggerGroup.leave()
```

```
    }
```

```
}
```

```
removalCharacteristicEvents.removeAll()
```

```
}
```

```

/// All HMCharacteristics in the targetValueMap.

private var allCharacteristics: [HMCharacteristic] {

    var characteristics = Set<HMCharacteristic>()

    for characteristic in targetValueMap.keyEnumerator().allObjects as!
[HMCharacteristic] {
        characteristics.insert(characteristic)
    }

    return Array(characteristics)
}

/**

```

Saves a characteristic and value into the pending map

of characteristic events.

- parameter value: The value of the characteristic.

- parameter characteristic: The HMCharacteristic that has been updated.

*/

private func updateEventValue(_ value: NSCopying, forCharacteristic
characteristic:

HMCharacteristic) {

for (index, event) in removalCharacteristicEvents.enumerated() {

if event.characteristic == characteristic {

/*

We have this event pending for deletion,

but we are going to want to update it.

remove it from the removal array.

*/

removalCharacteristicEvents.remove(at: index)

break

}

}

targetValueMap.setObject(value, forKey: characteristic)

}

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

/**

The current, sorted collection of HMCharacteristicEvents accumulated by filtering out the events pending removal from the original trigger events and then adding new pending events.

*/

```
var events: [HMCharacteristicEvent<CellValueType>] {  
    let characteristicEvents = eventTrigger?.characteristicEvents ?? []  
    let originalEvents = characteristicEvents.filter {  
        return !removalCharacteristicEvents.contains($0)  
    }  
}
```

let allEvents = originalEvents + pendingCharacteristicEvents

return allEvents.sorted { (event1: HMCharacteristicEvent, event2:

HMCharacteristicEvent) in

let type1 = event1.characteristic.localizedCharacteristicType

```

    let type2 = event2.characteristic.localizedCharacteristicType

    return type1.localizedCompare(type2) == .orderedAscending

}

}

// MARK: CharacteristicCellDelegate Methods

/**
If the mode is event, update the event value.
Otherwise, default to super implementation
*/

override func characteristicCell(_ cell: CharacteristicCell, didUpdateValue value:
CellValueType, forCharacteristic characteristic: HMCharacteristic, immediate:
Bool) {

    switch mode {

    case .event:

        updateEventValue(value, forCharacteristic: characteristic)

```

default:

```
super.characteristicCell(cell, didUpdateValue: value, forCharacteristic:
characteristic, immediate: immediate)

}

}

/**
Tries to find the characteristic in either the event map or the
condition map (based on the current mode). Then calls read value.
When the value comes back, we check the selected map for the value
*/

override func characteristicCell(_ cell: CharacteristicCell,
readInitialValueForCharacteristic characteristic: HMCharacteristic, completion:
@escaping
(CellValueType?, Error?) -> Void) {

    if mode == .condition {

        // This is a condition, fall back to the EventTriggerCreator read.
```



```

        super.characteristicCell(cell, readInitialValueForCharacteristic:
characteristic,
completion: completion)

```

```

        return

```

```

    }

```

```

    if let value = targetValueMap.object(forKey: characteristic) {

```

```

        completion(value, nil)

```

```

        return
    }

```

```

    characteristic.readValue { error in

```

```

        /*

```

The user may have updated the cell value while the

read was happening. We check the map one more time.

```

        */

```

```

    if let value = self.targetValueMap.object(forKey: characteristic) {

```

```

        completion(value, nil)
    }

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

    }

    else {

        completion(characteristic.value as? CellValueType, error)

    }

}

}

}

}

CharacteristicSelectionViewController

import UIKit

import HomeKit

/**

Allows for the selection of characteristics.

This is mainly used for creating characteristic events and conditions

*/

class CharacteristicSelectionViewController: HMCatalogViewController {

```

// MARK: Types

```
struct Identifiers {

    static let accessoryCell = "AccessoryCell"

    static let unreachableAccessoryCell = "UnreachableAccessoryCell"

    static let showServicesSegue = "Show Services"

}

// MARK: Properties

var eventTrigger: HMEventTrigger?

var triggerCreator: EventTriggerCreator!

/// An internal copy of all controllable accessories in the home.

private var accessories = [HMAccessory]()

@IBOutlet weak var saveButton: UIBarButtonItem!
```

```
// MARK: View Methods
```

```

/// Resets the internal array of accessories from the home.

override func viewWillAppear(_ animated: Bool) {

    super.viewWillAppear(animated)

    // Only take accessories which have one control service.

    accessories = home.sortedControlAccessories
}

/// Configures the ServicesViewController and passes it the correct accessory.

override func prepare(for segue: UIStoryboardSegue, sender: Any?) {

    if segue.identifier == Identifiers.showServicesSegue {

        let senderCell = sender as! UITableViewCell

        let servicesVC = segue.intendedDestinationViewController as!
ServicesViewController

        let cellIndex = tableView.indexPath(for: senderCell)!.row

        servicesVC.allowsAllWrites = true

        servicesVC.onlyShowsControlServices = true

        servicesVC.accessory = accessories[cellIndex]

```

```

servicesVC.cellDelegate = triggerCreator

}

}

```

```
// MARK: IBAction Methods
```

```
/**
```

Updates the predicates in the trigger creator and then dismisses the view controller.

```
*/
```

```
@IBAction func didTapSave(_ sender: UIBarButtonItem) {
```

```
/*
```

We should not save the trigger completely, the user still has a chance to bail out.

Instead, we generate all of the predicates that were in the map.

```
*/
```

```
triggerCreator.updatePredicates()
```

```
dismiss(animated: true, completion: nil)
```

```

}

// MARK: Table View Methods

/// Single section view controller.

override func numberOfSections(in tableView: UITableView) -> Int {

    return 1

}

/// - returns: The number of accessories. If there are none, will return 1 (for the
'none row').

override func tableView(_ tableView: UITableView, numberOfRowsInSectionSection
section: Int) -> Int {

    return max(accessories.count, 1)

}

/// - returns: An Accessory cell that contains an accessory's name.

override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) ->

```



```

UITableViewCell {

    let accessory = accessories.sortByLocalizedName()[indexPath.row]

    let cellIdentifier = accessory.isReachable ? Identifiers.accessoryCell :
Identifiers.unreachableAccessoryCell

    let cell = tableView.dequeueReusableCell(withIdentifier: cellIdentifier, for:
indexPath)

    cell.textLabel?.text = accessory.name

    return cell
}

/// Shows the services in the selected accessory.

override func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {

    tableView.deselectRow(at: indexPath, animated: true)

    let cell = tableView.cellForRow(at: indexPath)!

    if cell.selectionStyle == .none {

```

```

        return

    }

    performSegue(withIdentifier: Identifiers.showServicesSegue, sender: cell)

}

/// - returns: Localized "Accessories" string.

override func tableView(_ tableView: UITableView, titleForHeaderInSection
section: Int) ->
String? {
    return NSLocalizedString("Accessories", comment: "Accessories")
}
}

CharacteristicTriggerViewConTroller

/*

```

Copyright (C) 2016 Apple Inc. All Rights Reserved.

See LICENSE.txt for this sample's licensing information

Abstract:

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

The CharacteristicTriggerViewController allows the user to create a characteristic trigger.

*/

import UIKit

import HomeKit

/// A view controller which facilitates the creation of characteristic triggers.

class CharacteristicTriggerViewController: EventTriggerViewController {

// MARK: Types

struct Identifiers {

static let selectCharacteristicSegue = "Select Characteristic"

static let addCell = "AddCell"

static let conditionCell = "ConditionCell"

static let showTimeConditionSegue = "Show Time Condition"

}

```
// MARK: Properties
```

```
private var characteristicTriggerCreator: CharacteristicTriggerCreator {  
  
    return triggerCreator as! CharacteristicTriggerCreator  
  
}
```

```
var eventTrigger: HMEventTrigger? {  
  
    return trigger as? HMEventTrigger  
  
}
```

```
/// An internal array of HMCharacteristicEvents to save into the trigger.
```

```
private var events = [HMCharacteristicEvent<CellValueType>]()
```

```
// MARK: View Methods
```

```
/// Creates the trigger creator.
```

```

override func viewDidLoad() {

    super.viewDidLoad()

    triggerCreator = CharacteristicTriggerCreator(trigger: eventTrigger, home:
home)

}

/// Reloads the internal data.
override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)
    reloadData()
}

/// Passes our event trigger and trigger creator to the
CharacteristicSelectionViewController

override func prepare(for segue: UIStoryboardSegue, sender: Any?) {

    super.prepare(for: segue, sender: sender)

    if segue.identifier == Identifiers.selectCharacteristicSegue {

        if let destinationVC = segue.intendedDestinationViewController as?

```

```

CharacteristicSelectionViewController {

    destinationVC.eventTrigger = eventTrigger

    destinationVC.triggerCreator = characteristicTriggerCreator

}

}

}

// MARK: Table View Methods

/**
- returns: The characteristic events for the Characteristics section.
Defaults to super implementation.

*/

override func tableView(_ tableView: UITableView, numberOfRowsInSectionSection: Int) -> Int {

    switch sectionForIndex(section) {

        case .characteristics?:

            // Plus one for the add row.

```



```
return events.count + 1
```

```
case nil:
```

```
fatalError("Unexpected TriggerTableViewSection raw value.")
```

```
default:
```

```
return super.tableView(tableView, numberOfRowsInSection: section)
```

```
}
```

```
}
```

```
/**
```

Switches based on cell type to generate the correct cell for the index path.

Defaults to super implementation.

```
*/
```

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
```

```
UITableViewCell {
```

```
if indexPathIsAdd(indexPath) {
```

```

return self.tableView(tableView, addCellForRowAtIndexPath: indexPath)

}

switch sectionForIndex(indexPath.section) {

case .characteristics?:

return self.tableView(tableView, conditionCellForRowAtIndexPath:
indexPath)

case nil:

fatalError("Unexpected TriggerTableViewSection raw value.")

default:

return super.tableView(tableView, cellForRowAt: indexPath)

}

}

}

```

/// - returns: A 'condition cell' with the event at the specified index path.

```
private func tableView(_ tableView: UITableView,
conditionCellForRowAtIndexPath indexPath:
```

```
IndexPath) -> UITableViewCell {
```

```
    let cell = tableView.dequeueReusableCell(withIdentifier:
Identifiers.conditionCell, for:
```

```
indexPath) as! ConditionCell
```

```
    let event = events[indexPath.row]
```

```
    cell.setCharacteristic(event.characteristic, targetValue: event.triggerValue!)
```

```
    return cell
```

```
}
```

```
/**
```

```
- returns: An 'add cell' with localized text.
```

```
Defaults to super implementation.
```

```
*/
```

```
override func tableView(_ tableView: UITableView, addCellForRowAtIndexPath
indexPath:
```

```
IndexPath) -> UITableViewCell {
```

```
    switch sectionForIndex(indexPath.section) {
```

```

        case .characteristics?:

            let cell = tableView.dequeueReusableCell(withIdentifier:
Identifiers.addCell, for:
indexPath)

            cell.textLabel?.text = NSLocalizedString("Add Characteristic...",
comment: "Add
Characteristic")

            cell.textLabel?.textColor = .editableBlue

            return cell

        case nil:

            fatalError("Unexpected TriggerTableViewSection raw value.")

        default:

            return super.tableView(tableView, addCellForRowAtIndexPath:
indexPath)

    }

}

```

```
/**
```

Handles the selection of characteristic events.

Defaults to super implementation for other sections.

```
*/
```

```
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
```

```
    tableView.deselectRow(at: indexPath, animated: true)
```

```
    switch sectionForIndex(indexPath.section) {
```

```
        case .characteristics?:
```

```
            if indexPathIsAdd(indexPath) {
```

```
                addEvent()
```

```
                return
```

```
            }
```

```
            let cell = tableView.cellForRow(at: indexPath)
```

```
            performSegue(withIdentifier: Identifiers.selectCharacteristicSegue, sender: cell)
```

```
        case nil:
```

```
fatalError("Unexpected TriggerTableViewSection raw value.")
```

default:

```
super.tableView(tableView, didSelectRowAt: indexPath)
```

```
}
```

```
}
```

```
/**
```

- returns: true for characteristic cells,

otherwise defaults to super implementation.

```
*/
```

```
override func tableView(_ tableView: UITableView, canEditRowAt indexPath: IndexPath) -> Bool
```

```
{
```

```
    if indexPathIsAdd(indexPath) {
```

```
        return false
```

```
    }
```

```
    switch sectionForIndex(indexPath.section) {
```



```
case .characteristics?:
```

```
    return true
```

```
case nil:
```

```
    fatalError("Unexpected TriggerTableViewSection raw value.")
```

```
default:
```

```
    return super.tableView(tableView, canEditRowAt: indexPath)
```

```
}
```

```
}
```

```
/**
```

Removes events from the trigger creator.

Defaults to super implementation for other sections.

```
*/
```

```
override func tableView(_ tableView: UITableView, commit editingStyle:
```

```
UITableViewCellEditingStyle, forRowAt indexPath: IndexPath) {
```

```
    if editingStyle == .delete {
```

```

switch sectionForIndex(indexPath.section) {

    case .characteristics?:

        characteristicTriggerCreator.removeEvent(events[indexPath.row])

        events = characteristicTriggerCreator.events

        tableView.deleteRows(at: [indexPath], with: .automatic)

    case nil:

        fatalError("Unexpected TriggerTableViewSection raw value.")

    default:

        super.tableView(tableView, commit: editingStyle, forRowAt:
indexPath)

}

}

}

/**

```

- returns: A localized description of characteristic events

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Defaults to super implementation for other sections.

```
*/
```

```
override func tableView(_ tableView: UITableView, titleForFooterInSection  
section: Int) ->
```

```
String? {
```

```
    switch sectionForIndex(section) {
```

```
        case .characteristics?:
```

```
            return NSLocalizedString("This trigger will activate when any of these  
characteristics
```

```
change to their value. For example, 'run when the garage door is opened'.", comment:
```

```
"Characteristic Trigger Description")
```

```
        case nil:
```

```
            fatalError("Unexpected TriggerTableViewSection raw value.")
```

```
    default:
```

```
        return super.tableView(tableView, titleForFooterInSection: section)
```

```
    }
```

```

}

// MARK: Helper Methods

/// Resets the internal events array from the trigger creator.

private func reloadData() {

    events = characteristicTriggerCreator.events

    tableView.reloadData()

}

/// Performs a segue to the CharacteristicSelectionViewController.

private func addEvent() {

    characteristicTriggerCreator.mode = .event

    self.performSegue(withIdentifier: Identifiers.selectCharacteristicSegue, sender:
nil)

}

```

```
/// - returns: true if the section is the Characteristic 'add row'; otherwise defaults  
to super
```

implementation.

```
override func indexPathIsAdd(_ indexPath: IndexPath) -> Bool {
```

```
    switch sectionForIndex(indexPath.section) {
```

```
        case .characteristics?:
```

```
            return indexPath.row == events.count
```

```
        case nil:
```

```
            fatalError("Unexpected TriggerTableViewSection raw value.")
```

```
        default:
```

```
            return super(indexPathIsAdd(indexPath))
```

```
    }
```

```
}
```

```
// MARK: Trigger Controller Methods
```

```
/**
```

- parameter index: The section index.

- returns: The TriggerTableViewSection for the given index.

```
*/
```

```
override func sectionForIndex(_ index: Int) -> TriggerTableViewSection? {
```

```
    switch index {
```

```
        case 0:
```

```
            return .name
```

```
        case 1:
```

```
            return .enabled
```

```
        case 2:
```

```
            return .characteristics
```

```
        case 3:
```

```
            return .conditions
```


case 4:

return .actionSets

default:

return nil

}

}

}

MapViewController

import UIKit

import MapKit

/**

Allows the sender to get notified when there

have been changes to the region.

*/

```

protocol MapViewControllerDelegate {

    /**

        Notifies the delegate that the MapViewController's

        region has been updated.

    */

    func mapViewDidUpdateRegion(_ region: CLCircularRegion)
}

/**

    A view controller which allows the selection of a

    circular region on a map.

*/

class MapViewController: UIViewController, UISearchBarDelegate,
CLLocationManagerDelegate, MKMapViewDelegate {

    // MARK: Types

    struct Identifiers {

        static let circularRegion = "MapViewController.Region"

```

```

}

/// When the view loads, we'll zoom to this longitude/latitude span delta.

static let InitialZoomDelta: Double = 0.0015

/// When the view loads, we'll zoom into this span.

static let InitialZoomSpan = MKCoordinateSpan(latitudeDelta:
MapViewController.InitialZoomDelta, longitudeDelta:
MapViewController.InitialZoomDelta)

// The inverse of the percentage of the map view that should be captured in the
region.

static let MapRegionFraction: Double = 4.0

// The size of the query region with respect to the map's zoom.

static let RegionQueryDegreeMultiplier: Double = 5.0

// MARK: Properties

```

```
@IBOutlet weak var overlayView: MapOverlayView!
```

```
@IBOutlet weak var searchBar: UISearchBar!
```

```
@IBOutlet weak var mapView: MKMapView!
```

```
var delegate: MapViewControllerDelegate?
```

```
var targetRegion: CLCircularRegion?
```

```
var circleOverlay: MKCircle? {
```

```
    didSet {
```

```
        // Remove the old overlay (if exists)
```

```
        if let oldOverlay = oldValue {
```

```
            mapView.removeOverlay(oldOverlay)
```

```
        }
```

```
        // Add the new overlay (if exists)
```

```
        if let overlay = circleOverlay {
```

```
        mapView.addOverlay(overlay)

    }

}

}

var locationManager = CLLocationManager()

// MARK: View Methods

/// Configures the map view, search bar and location manager.
override func viewDidLoad() {

    super.viewDidLoad()

    searchBar.delegate = self

    mapView.delegate = self

    mapView.showsUserLocation = true

    mapView.isPitchEnabled = false

    locationManager.delegate = self

}
```

```
/// Loads the user's location and zooms the target region.
```

```
override func viewDidAppear(_ animated: Bool) {
```

```
    super.viewDidAppear(animated)
```

```
    locationManager.requestWhenInUseAuthorization()
```

```
    locationManager.requestLocation()
```

```
    if let region = targetRegion {
```

```
        annotateAndZoomToRegion(region)
```

```
    }
```

```
}
```

```
/// Updates the overlay when the orientation changes.
```

```
override func didRotate(from fromInterfaceOrientation: UIInterfaceOrientation) {
```

```
    overlayView.setNeedsDisplay()
```

```
}
```

```
// MARK: Button Actions
```



```
/**
```

Generates a map region based on the map's position

and zoom, then notifies the delegate that the region has changed.

This will dismiss the view.

```
*/
```

```
@IBAction func didTapSaveButton(_ sender: UIBarButtonItem) {
```

```
    let circleDegreeDelta: CLLocationDegrees
```

```
    let pointOnCircle: CLLocation
```

```
    if mapView.region.span.latitudeDelta > mapView.region.span.longitudeDelta {
```

```
        circleDegreeDelta = mapView.region.span.longitudeDelta /
```

```
        MapViewController.MapRegionFraction
```

```
        pointOnCircle = CLLocation(latitude: mapView.region.center.latitude,  
longitude:
```

```
mapView.region.center.longitude - circleDegreeDelta)
```

```
    }
```

```
    else {
```

```

        circleDegreeDelta = mapView.region.span.latitudeDelta /
MapViewController.MapRegionFraction

        pointOnCircle = CLLocation(latitude: mapView.region.center.latitude -
circleDegreeDelta, longitude: mapView.region.center.longitude)

    }

    let mapCenterLocation = CLLocation(latitude: mapView.region.center.latitude,
longitude: mapView.region.center.longitude)

    let distance = pointOnCircle.distance(from: mapCenterLocation)

    let genericRegion = CLCircularRegion(center: mapView.region.center, radius:
distance, identifier: Identifiers.circularRegion)

    circleOverlay = MKCircle(center: genericRegion.center, radius:
genericRegion.radius)

    delegate?.mapViewDidUpdateRegion(genericRegion)

    dismiss(animated: true, completion: nil)

}

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

/// Dismisses the view without notifying the delegate.

@IBAction func didTapCancelButton(_ sender: UIBarButtonItem) {

    dismiss(animated: true, completion: nil)

}

// MARK: Search Bar Methods

/**
Dismisses the keyboard and runs a new search from the
search bar.

*/

func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {

    searchBar.resignFirstResponder()

    mapView.removeAnnotations(mapView.annotations)

    performSearch()

}

```

```

// MARK: Location Manager Methods

/// Zooms to the user's location if the region is not set.

func locationManager(_ manager: CLLocationManager, didUpdateLocations
locations:
[CLLocation]) {

    guard let lastLocation = locations.last else { return }

    if targetRegion != nil {

        // Do not zoom to the user's location if there is already a target region.
        return
    }

    let newRegion = MKCoordinateRegion(center: lastLocation.coordinate, span:
MapViewController.InitialZoomSpan)

    mapView.setRegion(newRegion, animated: true)

}

/**

```

The method is required.

Simply logs the error.

```
*/
```

```
func locationManager(_ manager: CLLocationManager, didFailWithError error: Error) {
```

```
    print("System: Location Manager Error: \(error)")
```

```
}
```

```
/**
```

When the user updates the authorization status, we want to zoom to their current location by asking for it.

```
*/
```

```
func locationManager(_ manager: CLLocationManager, didChangeAuthorization status:
```

```
    CLAuthorizationStatus) {
```

```
        locationManager.requestLocation()
```

```
}
```

```
// MARK: Helper Methods
```

/**

Updates the region overlay and zooms the map region

- parameter region: The new CLCircularRegion.

*/

```
private func annotateAndZoomToRegion(_ region: CLCircularRegion) {  
    circleOverlay = MKCircle(center: region.center, radius: region.radius)  
    let multiplier = MapViewController.MapRegionFraction  
    let mapRegion = MKCoordinateRegion.init(center: region.center,  
latitudinalMeters:  
region.radius*multiplier, longitudinalMeters: region.radius*multiplier)  
    mapView.setRegion(mapRegion, animated: false)  
}
```

/**

Performs a natural language search for locations

in the map's region that match the searchBar's text.


```

*/

private func performSearch() {

    let request = MKLocalSearch.Request()

    request.naturalLanguageQuery = searchBar.text

    let multiplier = MapViewController.RegionQueryDegreeMultiplier

    let querySpan = MKCoordinateSpan(latitudeDelta:
mapView.region.span.latitudeDelta*multiplier, longitudeDelta:
mapView.region.span.longitudeDelta*multiplier)

    request.region = MKCoordinateRegion(center: mapView.region.center, span:
querySpan)

    let search = MKLocalSearch(request: request)

    var matchingItems = [MKMapItem]()

    search.start { response, error in

        let mapItems: [MKMapItem] = response?.mapItems ?? []

        for item in mapItems {

```

```

        matchingItems.append(item)

        let annotation = MKPointAnnotation()

        annotation.coordinate = item.placemark.coordinate

        annotation.title = item.name

        self.mapView.addAnnotation(annotation)

    }

}

}

/// - returns: An MKOverlayRenderer with our custom stroke and fill.
func mapView(_ mapView: MKMapView, rendererFor overlay: MKOverlay) ->
MKOverlayRenderer {

    let circleRenderer = MKCircleRenderer(overlay: overlay)

    circleRenderer.fillColor = UIColor.blue.withAlphaComponent(0.2)

    circleRenderer.strokeColor = UIColor.black

    circleRenderer.lineWidth = 2.0

    return circleRenderer

}

```

```
}
```

```
MapOverlayView
```

```
import MapKit
```

```
/**
```

A simple UIView subclass to draw a selection circle over
a MKMapView of the same size.

```
*/
```

```
class MapOverlayView: UIView {
```

```
/**
```

Draws a dashed circle in the center of the rect with
a radius 1/4th of the rect's smallest side.

```
*/
```

```
override func draw(_ rect: CGRect) {
```

```
    super.draw(rect)
```

```
    let context = UIGraphicsGetCurrentContext()
```

```

let strokeColor = UIColor.blue

let circleDiameter: CGFloat = min(rect.width, rect.height) / 2.0

let circleRadius = circleDiameter / 2.0

let circleRect = CGRect(x: rect.midX - circleRadius, y: rect.midY -
circleRadius, width:
circleDiameter, height: circleDiameter)

let circlePath = UIBezierPath(ovalIn: circleRect)

strokeColor.setStroke()

circlePath.lineWidth = 3

context!.saveGState()

context!.setLineDash(phase: 0, lengths: [6, 6])

circlePath.stroke()

context!.restoreGState()

}

/// - returns: false to accept no touches.

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

override func point(inside point: CGPoint, with event: UIEvent?) -> Bool {

    return false

}

}

```

LocationTriggerCreator

```
import HomeKit
```

```
import MapKit
```

```
/**
```

An EventTriggerCreator subclass which allows for the creation
of location triggers.

```
*/
```

```
class LocationTriggerCreator: EventTriggerCreator, MapViewControllerDelegate {
```

```
// MARK: Properties
```

```
var eventTrigger: HMEventTrigger? {
```

```
    return trigger as? HMEventTrigger
```

```
}
```

```

var locationEvent: HMLocationEvent?

var targetRegion: CLCircularRegion?

var targetRegionStateIndex = 0


// MARK: Trigger Creator Methods


/// Initializes location event, target region, and region state.
required init(trigger: HMTrigger?, home: HMHome) {
    super.init(trigger: trigger, home: home)
    if let eventTrigger = eventTrigger {
        self.locationEvent = eventTrigger.locationEvent
        if let region = locationEvent?.region as? CLCircularRegion {
            self.targetRegion = region
        }

        self.targetRegionStateIndex = (self.targetRegion?.notifyOnEntry ?? true) ? 0
: 1

    }
}

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้


```

}

/// Generates a new region and updates the location event.

override func updateTrigger() {

    if let region = targetRegion {

        prepareRegion()

        if let locationEvent = locationEvent {

            saveTriggerGroup.enter()

            locationEvent.updateRegion(region) { error in

                if let error = error {

                    self.errors.append(error)

                }

                self.saveTriggerGroup.leave()

            }

        }

    }

}

self.savePredicate()

```

```
}
```

```
/**
```

- returns: A new HMEventTrigger with a new generated

location event and predicate.

```
*/
```

```
override func newTrigger() -> HMTrigger? {
```

```
var events = [HMLocationEvent]()
```

```
if let region = targetRegion {
```

```
    prepareRegion()
```

```
    events.append(HMLocationEvent(region: region))
```

```
}
```

```
return HMEventTrigger(name: name, events: events, predicate: newPredicate())
```

```
}
```

```
// MARK: Helper Methods
```

```
/**
```

Sets the notifyOnEntry and notifyOnExit region

properties based on the selected state.

*/

```
private func prepareRegion() {
```

```
    if let region = targetRegion {
```

```
        region.notifyOnEntry = (targetRegionStateIndex == 0)
```

```
        region.notifyOnExit = !region.notifyOnEntry
```

```
    }
```

```
}
```

/**

Updates the target region from the one provided

by the delegate.

- parameter region: A new CLCircularRegion, provided by the delegate.

*/

```
func mapViewDidUpdateRegion(_ region: CLCircularRegion) {
```

```
    targetRegion = region
```

```

    }

}

LocationTriggerViewController

import UIKit

import MapKit

import HomeKit

import AddressBookUI

import Contacts

/// A view controller which facilitates the creation of a location trigger.
class LocationTriggerViewController: EventTriggerViewController {

    struct Identifiers {

        static let locationCell = "LocationCell"

        static let regionStatusCell = "RegionStatusCell"

        static let selectLocationSegue = "Select Location"

    }

```

```
static let geocoder = CLGeocoder()
```

```
static let regionStatusTitles = [
```

```
    NSLocalizedString("When I Enter The Area", comment: "When I Enter The  
Area"),
```

```
    NSLocalizedString("When I Leave The Area", comment: "When I Leave The  
Area")
```

```
]
```

```
var locationTriggerCreator: LocationTriggerCreator {  
    return triggerCreator as! LocationTriggerCreator  
}
```

```
var localizedAddress: String?
```

```
var viewIsDisplayed = false
```

```
// MARK: View Methods
```

```

/// Initializes a trigger creator and registers for table view cells.

override func viewDidLoad() {

    super.viewDidLoad()

    triggerCreator = LocationTriggerCreator(trigger: trigger, home: home)

    tableView.register(UITableViewCell.self, forCellReuseIdentifier:
Identifiers.locationCell)

    tableView.register(UITableViewCell.self, forCellReuseIdentifier:
Identifiers.regionStatusCell)
}

/**
Generates an address string for the current region location and
reloads the table view.

*/

override func viewWillAppear(_ animated: Bool) {

    super.viewWillAppear(animated)

    viewIsDisplayed = true

```



```

if let region = locationTriggerCreator.targetRegion {

    let centerLocation = CLLocation(latitude: region.center.latitude, longitude:
region.center.longitude)

    LocationTriggerViewController.geocoder.reverseGeocodeLocation(centerLocation)
    {

    placemarks, error in

        if !self.viewIsDisplayed {

            // The geocoder took too long, we're not on this view any more.
            return
        }

        if let error = error {

            self.displayError(error)

            return
        }

        if let mostLikelyPlacemark = placemarks?.first {

            let address = CNMutablePostalAddress(placemark:
mostLikelyPlacemark)

            let addressFormatter = CNPostalAddressFormatter()

```

```

        let addressString = addressFormatter.string(from: address)

        self.localizedAddress = addressString.replacingOccurrences(of: "\n",
with: ", ")

        let section = IndexSet(integer: 2)

        self.tableView.reloadSections(section, with: .automatic)

    }

}

tableView.reloadData()

}

/// Passes the trigger creator and region into the MapViewController.
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {

    super.prepare(for: segue, sender: sender)

    if segue.identifier == Identifiers.selectLocationSegue {

        guard let destinationVC = segue.intendedDestinationViewController as?
MapViewController else { return }

        // Give the map the previous target region (if exists).

```

```
destinationVC.targetRegion = locationTriggerCreator.targetRegion
```

```
destinationVC.delegate = locationTriggerCreator
```

```
}
```

```
}
```

```
override func viewWillDisappear(_ animated: Bool) {
```

```
    super.viewWillDisappear(animated)
```

```
    viewIsDisplayed = false
```

```
}
```

```
// MARK: Table View Methods
```

```
/**
```

```
- returns: The number of rows in the Region section;
```

```
defaults to the super implementation for other sections.
```

```
*/
```

```
override func tableView(_ tableView: UITableView, numberOfRowsInSectionSection  
section: Int) -> Int {
```

```

switch sectionForIndex(section) {

    case .region?:

        return 2

    case nil:

        fatalError("Unexpected TriggerTableViewSection raw value.")

    default:

        return super.tableView(tableView, numberOfRowsInSection: section)
}

}

/**

Generates a cell based on the section.

Handles Region and Location sections, defaults to

super implementations for other sections.

*/

```

```

    override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) ->

UITableViewCell {

    switch sectionForIndex(indexPath.section) {

        case .region?:

            return self.tableView(tableView, regionStatusCellForRowAtIndexPath:
indexPath)

        case .location?:

            return self.tableView(tableView, locationCellForRowAtIndexPath:
indexPath)

        case nil:

            fatalError("Unexpected TriggerTableViewSection raw value.")

        default:

            return super.tableView(tableView, cellForRowAt: indexPath)

    }

}

```

```

/// Generates the single location cell.

private func tableView(_ tableView: UITableView,
locationCellForRowAtIndexPath indexPath:
IndexPath) -> UITableViewCell {

    let cell = tableView.dequeueReusableCell(withIdentifier:
Identifiers.locationCell, for:
indexPath)

    cell.accessoryType = .disclosureIndicator

    if locationTriggerCreator.targetRegion != nil {

        cell.textLabel?.text = localizedAddress ?? NSLocalizedString("Update
Location", comment:
"Update Location")

    }

    else {

        cell.textLabel?.text = NSLocalizedString("Set Location", comment: "Set
Location")

    }
}

```



```

        return cell

    }

    /// Generates the cell which allow the user to select either 'on enter' or 'on exit'.

    private func tableView(_ tableView: UITableView,
regionStatusCellForRowAtIndexPath

indexPath: IndexPath) -> UITableViewCell {

        let cell = tableView.dequeueReusableCell(withIdentifier:
Identifiers.regionStatusCell, for:
indexPath)

        cell.textLabel?.text =
LocationTriggerViewController.regionStatusTitles[indexPath.row]

        cell.accessoryType = (locationTriggerCreator.targetRegionStateIndex ==
indexPath.row) ?

        .checkmark : .none

        return cell

    }

    /**

```

Allows the user to select a location or change the region status.

Defaults to the super implementation for other sections.

*/

```
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
```

```
    switch sectionForIndex(indexPath.section) {
```

```
        case .location?:
```

```
            performSegue(withIdentifier: Identifiers.selectLocationSegue, sender: self)
```

```
        case .region?:
```

```
            locationTriggerCreator.targetRegionStateIndex = indexPath.row
```

```
            let reloadIndexSet = IndexSet(integer: indexPath.section)
```

```
            tableView.reloadSections(reloadIndexSet, with: .automatic)
```

```
        case nil:
```

```
            fatalError("Unexpected TriggerTableViewSection raw value.")
```

default:

```
super.tableView(tableView, didSelectRowAt: indexPath)
```

```
}
```

```
}
```

```
/**
```

- returns: A localized title for the Location and Region sections.

Defaults to the super implementation for other sections.

```
*/
```

```
override func tableView(_ tableView: UITableView, titleForHeaderInSection  
section: Int) ->
```

```
String? {
```

```
switch sectionForIndex(section) {
```

```
case .location?:
```

```
return NSLocalizedString("Location", comment: "Location")
```

```
case .region?:
```

```
return NSLocalizedString("Region Status", comment: "Region Status")
```

case nil:

```
fatalError("Unexpected TriggerTableViewSection raw value.")
```

default:

```
return super.tableView(tableView, titleForHeaderInSection: section)
```

```
}
```

```
}
```

```
/**
```

- returns: A localized description of the region status.

Defaults to the super implementation for other sections.

```
*/
```

```
override func tableView(_ tableView: UITableView, titleForFooterInSection  
section: Int) ->
```

```
String? {
```

```
switch sectionForIndex(section) {
```

```
case .region?:
```

```
return NSLocalizedString("This trigger can activate when you enter or  
leave a region. For
```

```
example, when you arrive at home or when you leave work.", comment: "Location  
Region
```

```
Description")
```

```
case nil:
```

```
fatalError("Unexpected TriggerTableViewSection raw value.")
```

```
default:
```

```
return super.tableView(tableView, titleForFooterInSection: section)
```

```
}
```

```
}
```

```
// MARK: Trigger Controller Methods
```

```
/**
```

```
- parameter index: The section index.
```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 415 ไปถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

- returns: The TriggerTableViewSection for the given index.

*/

override func sectionForIndex(_ index: Int) -> TriggerTableViewSection? {

switch index {

case 0:

return .name

case 1:

return .enabled

case 2:

return .location

case 3:

return .region

case 4:


```
return .conditions
```

case 5:

```
return .actionSets
```

default:

```
return nil
```

```
}
```

```
}
```

```
}
```

```
EventTriggerCreator
```

```
import HomeKit
```

```
/**
```

A superclass for event trigger creators.

These classes manage the state for characteristic trigger conditions.

```
*/
```

```
class EventTriggerCreator: TriggerCreator, CharacteristicCellDelegate {
```

```
    // MARK: Properties
```

```
    /// A mapping of HMCharacteristics to their values.
```

```
    private let conditionValueMap = NSMapTable<HMCharacteristic,  
    CellValueType>.strongToStrongObjects()
```

```
    private var eventTrigger: HMEventTrigger? {  
        return trigger as? HMEventTrigger  
    }
```

```
    /**
```

An array of top-level NSPredicate objects.

Currently, HMCatalog only supports top-level NSPredicates

which have type AndPredicateType.

```
    */
```

```
    var originalConditions: [NSPredicate] {
```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า

ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 418 ังอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

        if let compoundPredicate = eventTrigger?.predicate as?
NSCompoundPredicate,

        let subpredicates = compoundPredicate.subpredicates as? [NSPredicate] {

            return subpredicates

        }

        return []
    }

    /// An array of new conditions which will be written when the trigger is saved.
    lazy var conditions: [NSPredicate] = self.originalConditions

    /**

    Adds a predicate to the pending conditions.

    - parameter predicate: The new NSPredicate to add.

    */

    func addCondition(_ predicate: NSPredicate) {

```

```

conditions.append(predicate)

}

/**

Removes a predicate from the pending conditions.

- parameter predicate: The NSPredicate to remove.

*/
func removeCondition(_ predicate: NSPredicate) {
    if let index = conditions.firstIndex(of: predicate) {
        conditions.remove(at: index)
    }
}

/**

- returns: The new NSCompoundPredicate, generated from

the pending conditions.

*/

```

```

func newPredicate() -> NSPredicate {

    return NSCompoundPredicate(type: .and, subpredicates: conditions)

}

/// Handles the value update and stores the value in the condition map.

func characteristicCell(_ cell: CharacteristicCell, didUpdateValue value:
CellValueType,
forCharacteristic characteristic: HMCharacteristic, immediate: Bool) {
    conditionValueMap.setObject(value, forKey: characteristic)
}

/**

Tries to use the value from the condition-value map, but falls back

to reading the characteristic's value from HomeKit.

*/

func characteristicCell(_ cell: CharacteristicCell,
readInitialValueForCharacteristic characteristic:
HMCharacteristic, completion: @escaping (CellValueType?, Error?) -> Void) {

```

```

if let value = conditionValueMap.object(forKey: characteristic) {

    completion(value, nil)

    return

}

```

```

characteristic.readValue { error in

```

```

/*

```

The user may have updated the cell value while the read was happening. We check the map one more time.

```

*/

```

```

if let value = self.conditionValueMap.object(forKey: characteristic) {

    completion(value, nil)

}

```

```

else {

```

```

    completion(characteristic.value as? CellValueType, error)

```

```

}

```

```

}

```

```

}

```



```
// MARK: Helper Methods
```

```
/**
```

Updates the predicates and saves the new, generated
predicate to the event trigger.

```
*/
```

```
func savePredicate() {  
    updatePredicates()  
    saveTriggerGroup.enter()  
    eventTrigger?.updatePredicate(newPredicate()) { error in  
        if let error = error {  
            self.errors.append(error)  
        }  
  
        self.saveTriggerGroup.leave()  
    }  
}
```

```
/// Generates predicates from the characteristic-value map and adds them to the pending
```

conditions.

```
func updatePredicates() {  
  
    for (characteristic, value) in pairsFromMapTable(conditionValueMap) {  
  
        let predicate =  
HMEventTrigger.predicateForEvaluatingTrigger(characteristic, relatedBy:  
.equalTo, toValue: value)  
        addCondition(predicate)  
    }  
  
    conditionValueMap.removeAllObjects()  
}  
  
/**
```

- parameter table: The NSMapTable from which to generate the pairs.

- returns: Tuples representing HMCharacteristics and their associated return trigger values.

```

*/

func pairsFromMapTable<Key, ValueType>(_ table: NSMapTable<Key,
ValueType>) -> [(Key,
ValueType)] {

    return table.keyEnumerator().allObjects.map { object in

        let characteristic = object as! Key

        let triggerValue = table.object(forKey: characteristic)

        return (characteristic, triggerValue!)

    }

}

ConditionCell

import UIKit

import HomeKit

/// A UITableViewCell subclass that displays a trigger condition.

class ConditionCell: UITableViewCell {

    /// A static, short date formatter.

```

```

static let dateFormatter: DateFormatter = {

    let dateFormatter = DateFormatter()

    dateFormatter.dateStyle = .none

    dateFormatter.timeStyle = .short

    dateFormatter.locale = Locale.current

    return dateFormatter

}()

/// Ignores the passed-in style and overrides it with .Subtitle.
override init(style: UITableViewCell.CellStyle, reuseIdentifier: String?) {

    super.init(style: .subtitle, reuseIdentifier: reuseIdentifier)

    selectionStyle = .none

    detailTextLabel?.textColor = UIColor.lightGray

    accessoryType = .none

}

/// Required because we overwrote a designated initializer.

required init?(coder aDecoder: NSCoder) {

```

```
super.init(coder: aDecoder)
```

```
}
```

```
/**
```

Sets the cell's text to represent a characteristic and target value.

For example, "Brightness → 60%"

Sets the subtitle to the service and accessory that this characteristic represents.

- parameter characteristic: The characteristic this cell represents.

- parameter targetValue: The target value from this action.

```
*/
```

```
func setCharacteristic(_ characteristic: HMCharacteristic, targetValue: AnyObject) {
```

```
    let targetDescription = "\(characteristic.localizedDescription) →
```

```
    \(characteristic.localizedDescriptionForValue(targetValue))"
```

```
    textLabel?.text = targetDescription
```

```

let contextDescription = NSLocalizedString("%@ in %@", comment: "Service
in Accessory")

if let service = characteristic.service, let accessory = service.accessory {

    detailTextLabel?.text = String(format: contextDescription, service.name,
accessory.name)

}

else {

    detailTextLabel?.text = NSLocalizedString("Unknown Characteristic",
comment: "Unknown
Characteristic")
}
}

/**

```

Sets the cell's text to represent an ordered time with a set context string.

- parameter order: A TimeConditionOrder which will map to a localized string.
- parameter timeString: The localized time string.
- parameter contextString: A localized string describing the time type.


```

*/

private func setOrder(_ order: TimeConditionOrder, timeString: String,
contextString: String) {

    let formatString: String

    switch order {

    case .before:

        formatString = NSLocalizedString("Before %@", comment: "Before
Time")

    case .after:

        formatString = NSLocalizedString("After %@", comment: "After Time")

    case .at:

        formatString = NSLocalizedString("At %@", comment: "At Time")

    }

    titleLabel?.text = String(format: formatString, timeString)

    detailTextLabel?.text = contextString

}

```

/**

Sets the cell's text to represent an exact time condition.

- parameter order: A `TimeConditionOrder` which will map to a localized string.
- parameter dateComponents: The date components of the exact time.

*/

```
func setOrder(_ order: TimeConditionOrder, dateComponents: DateComponents)
{
    let date = Calendar.current.date(from: dateComponents)
    let timeString = ConditionCell.dateFormatter.string(from: date!)

    setOrder(order, timeString: timeString, contextString:
    NSLocalizedString("Relative to Time",
    comment: "Relative to Time"))
}
```

/**

Sets the cell's text to represent a solar event time condition.

- parameter order: A TimeConditionOrder which will map to a localized string.
- parameter sunState: A TimeConditionSunState which will map to localized string.

*/

```
func setOrder(_ order: TimeConditionOrder, sunState: TimeConditionSunState) {

    let timeString: String

    switch sunState {
    case .sunrise:
        timeString = NSLocalizedString("Sunrise", comment: "Sunrise")

    case .sunset:
        timeString = NSLocalizedString("Sunset", comment: "Sunset")

    }

    setOrder(order, timeString: timeString, contextString:
NSLocalizedString("Relative to sun",
comment: "Relative to Sun"))

}
```

```

/// Sets the cell's text to indicate the given condition is not handled by the app.

func setUnknown() {

    let unknownString = NSLocalizedString("Unknown Condition", comment:
"Unknown
Condition")

    detailTextLabel?.text = unknownString
   .textLabel?.text = unknownString
}
}

import UIKit

/// A UITableViewCell subclass with a UIDatePicker, used for selecting a specific
time of
day.

class TimePickerCell: UITableViewCell {

    // MARK: Properties

```

```

@IBOutlet weak var datePicker: UIDatePicker!

}

SegmentedTimeCell

import UIKit

/// A UITableViewCell subclass with a UISegmentedControl, used for selecting the
time type.

class SegmentedTimeCell: UITableViewCell {

    // MARK: Properties

    @IBOutlet weak var segmentedControl: UISegmentedControl!

}

TimeConditionViewController

import UIKit

import HomeKit

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 433 ไปถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

/// Represents a section in the TimeConditionViewController.

```
enum TimeConditionTableViewSection: Int {
```

```
/**
```

This section contains the segmented control to
choose a time condition type.

```
*/
```

```
case timeOrSun
```

```
/**
```

This section contains cells to allow the selection
of 'before', 'after', or 'at'. 'At' is only available
when the exact time is specified.

```
*/
```

```
case beforeOrAfter
```

```
/**
```

If the condition type is exact time, this section will

only have one cell, the date picker cell.

If the condition type is relative to a solar event,

this section will have two cells, one for 'sunrise' and

one for 'sunset.

*/

case value

static let count = 3

}

/**

Represents the type of time condition.

The condition can be an exact time, or relative to a solar event.

*/

enum TimeConditionType: Int {

case time, sun

```
}
```

```
/**
```

Represents the type of solar event.

This can be sunrise or sunset.

```
*/
```

```
enum TimeConditionSunState: Int {
```

```
case sunrise, sunset
```

```
}
```

```
/**
```

Represents the condition order.

Conditions can be before, after, or exactly at a given time.

```
*/
```

```
enum TimeConditionOrder: Int {
```

```
case before, after, at
```

```

}

/// A view controller that facilitates the creation of time conditions for triggers.

class TimeConditionViewController: HMCatalogViewController {

    // MARK: Types

    struct Identifiers {

        static let selectionCell = "SelectionCell"

        static let timePickerCell = "TimePickerCell"

        static let segmentedTimeCell = "SegmentedTimeCell"

    }

    static let timeOrSunTitles = [

        NSLocalizedString("Relative to time", comment: "Relative to time"),

        NSLocalizedString("Relative to sun", comment: "Relative to sun")

    ]

    static let beforeOrAfterTitles = [

```

```
    NSLocalizedString("Before", comment: "Before"),

    NSLocalizedString("After", comment: "After"),

    NSLocalizedString("At", comment: "At")

]
```

```
static let sunriseSunsetTitles = [

    NSLocalizedString("Sunrise", comment: "Sunrise"),

    NSLocalizedString("Sunset", comment: "Sunset")

]
```

```
// MARK: Properties
```

```
private var timeType: TimeConditionType = .time
```

```
private var order: TimeConditionOrder = .before
```

```
private var sunState: TimeConditionSunState = .sunrise
```

```
private var datePicker: UIDatePicker?
```

```
var triggerCreator: EventTriggerCreator?
```

```
// MARK: View Methods
```

```
/// Configures the table view.
```

```
override func viewDidLoad() {
```

```
    super.viewDidLoad()
```

```
    tableView.rowHeight = UITableView.automaticDimension
```

```
    tableView.estimatedRowHeight = 44.0
```

```
}
```

```
// MARK: Table View Methods
```

```
/// - returns: The number of TimeConditionTableViewSections.
```

```
override func numberOfSections(in tableView: UITableView) -> Int {
```

```
    return TimeConditionTableViewSection.count
```

```
}
```

```
/**
```

- returns: The number rows based on the TimeConditionTableViewSection
and the timeType.

```
*/
```

```
override func tableView(_ tableView: UITableView, numberOfRowsInSectionSection: Int) -> Int {
```

```
    switch TimeConditionTableViewSection(rawValue: section) {
```

```
        case .timeOrSun?:
```

```
            return 1
```

```
        case .beforeOrAfter?:
```

```
            // If we're choosing an exact time, we add the 'At' row.
```

```
            return (timeType == .time) ? 3 : 2
```

```
        case .value?:
```

```
            // Date picker cell or sunrise/sunset selection cells
```

```
            return (timeType == .time) ? 1 : 2
```



```

case nil:

    fatalError("Unexpected TimeConditionTableViewSection raw value.")

}

}

/// Switches based on the section to generate a cell.

override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) ->
UITableViewCell {

    switch TimeConditionTableViewSection(rawValue: indexPath.section) {

        case .timeOrSun?:

            return self.tableView(tableView, segmentedCellForRowAtIndexPath:
indexPath)

        case .beforeOrAfter?:

            return self.tableView(tableView, selectionCellForRowAtIndexPath:
indexPath)

        case .value?:

```

```

switch timeType {

    case .time:

        return self.tableView(tableView, datePickerCellForRowAtIndexPath:
indexPath)

    case .sun:

        return self.tableView(tableView, selectionCellForRowAtIndexPath:
indexPath)

    }

    case nil:

        fatalError("Unexpected TimeConditionTableViewSection raw value.")

    }

}

/// - returns: A localized string describing the section.

override func tableView(_ tableView: UITableView, titleForHeaderInSection
section: Int) ->

String? {

    switch TimeConditionTableViewSection(rawValue: section) {

```

```
case .timeOrSun?:
```

```
    return NSLocalizedString("Condition Type", comment: "Condition Type")
```

```
case .beforeOrAfter?:
```

```
    return nil
```

```
case .value?:
```

```
    if timeType == .time {
```

```
        return NSLocalizedString("Time", comment: "Time")
```

```
    }
```

```
    else {
```

```
        return NSLocalizedString("Event", comment: "Event")
```

```
    }
```

```
case nil:
```

```
    fatalError("Unexpected TimeConditionTableViewSection raw value.")
```

```
}
```

```
}
```

```

/// - returns: A localized description for condition type section; nil otherwise.

override func tableView(_ tableView: UITableView, titleForFooterInSection
section: Int) ->

String? {

    switch TimeConditionTableViewSection(rawValue: section) {

        case .timeOrSun?:

            return NSLocalizedString("Time conditions can relate to specific times or
special events,
like sunrise and sunset.", comment: "Condition Type Description")

        case .beforeOrAfter?:

            return nil

        case .value?:

            return nil

        case nil:

```

```

        fatalError("Unexpected TimeConditionTableViewSection raw value.")

    }

}

/// Updates internal values based on row selection.

override func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {

    let cell = tableView.cellForRow(at: indexPath)!

    if cell.selectionStyle == .none {

        return

    }

    tableView.deselectRow(at: indexPath, animated: true)

    switch TimeConditionTableViewSection(rawValue: indexPath.section) {

    case .timeOrSun?:

        timeType = TimeConditionType(rawValue: indexPath.row)!

        reloadDataDynamicSections()

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 445 ังอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

return

case .beforeOrAfter?:

    order = TimeConditionOrder(rawValue: indexPath.row)!

    tableView.reloadSections(IndexSet(integer: indexPath.section), with:
.automatic)

case .value?:

    if timeType == .sun {

        sunState = TimeConditionSunState(rawValue: indexPath.row)!

    }

    tableView.reloadSections(IndexSet(integer: indexPath.section), with:
.automatic)

case nil:

    fatalError("Unexpected TimeConditionTableViewSection raw value.")

}

}

```



```

// MARK: Helper Methods

/**
Generates a selection cell based on the section.
Ordering and sun-state sections have selections.
*/
private func tableView(_ tableView: UITableView,
selectionCellForRowAtIndexPath indexPath:
IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier:
Identifiers.selectionCell, for:
indexPath)

    switch TimeConditionTableViewSection(rawValue: indexPath.section) {

        case .beforeOrAfter?:

            cell.textLabel?.text =
TimeConditionViewController.beforeOrAfterTitles[indexPath.row]

            cell.accessoryType = (order.rawValue == indexPath.row) ? .checkmark :
.none

```

```

case .value?:

    if timeType == .sun {

        cell.textLabel?.text =
TimeConditionViewController.sunriseSunsetTitles[indexPath.row]

        cell.accessoryType = (sunState.rawValue == indexPath.row) ?
.checkmark : .none

    }

case nil:
    fatalError("Unexpected TimeConditionTableViewSection raw value.")

default:
    break

}

return cell

}

```

```

/// Generates a date picker cell and sets the internal date picker when created.

private func tableView(_ tableView: UITableView,
datePickerCellForRowAtIndexPath indexPath:
IndexPath) -> UITableViewCell {

    let cell = tableView.dequeueReusableCell(withIdentifier:
Identifiers.timePickerCell, for:
indexPath) as! TimePickerCell

    // Save the date picker so we can get the result later.
    datePicker = cell.datePicker
    return cell
}

/// Generates a segmented cell and sets its target when created.

private func tableView(_ tableView: UITableView,
segmentedCellForRowAtIndexPath indexPath:
IndexPath) -> UITableViewCell {

    let cell = tableView.dequeueReusableCell(withIdentifier:
Identifiers.segmentedTimeCell, for:
indexPath) as! SegmentedTimeCell

```

```

cell.segmentedControl.selectedSegmentIndex = timeType.rawValue

cell.segmentedControl.removeTarget(nil, action: nil, for: .allEvents)

cell.segmentedControl.addTarget(self, action:

#selector(self.segmentedControlDidChange(_:)), for: .valueChanged)

return cell

}

/// Creates date components from the date picker's date.
var dateComponents: DateComponents? {
    guard let datePicker = datePicker else { return nil }
    let flags: Set<Calendar.Component> = [.hour, .minute]
    return Calendar.current.dateComponents(flags, from: datePicker.date)
}

/**

Updates the time type and reloads dynamic sections.

```

- parameter segmentedControl: The segmented control that changed.

```

*/

@objc func segmentedControlDidChange(_ segmentedControl:
UISegmentedControl) {

    if let segmentedControlType = TimeConditionType(rawValue:
segmentedControl.selectedSegmentIndex) {

        timeType = segmentedControlType
    }

    reloadDynamicSections()
}

/// Reloads the BeforeOrAfter and Value section.

private func reloadDynamicSections() {

    if timeType == .sun && order == .at {

        order = .before
    }

    let reloadIndexSet = IndexSet(integersIn:
Range(NSMakeRange(TimeConditionTableViewSection.beforeOrAfter.rawValue,
2)) ?? 0..<0)

```

```

tableView.reloadSections(reloadIndexSet, with: .automatic)

}

// MARK: IBAction Methods

/**
Generates a predicate based on the stored values, adds
the condition to the trigger, then dismisses the view.
*/
@IBAction func saveAndDismiss(_ sender: UIBarButtonItem) {
    var predicate: NSPredicate?

    switch timeType {

    case .time:

        switch order {

        case .before:

            predicate
            HMEventTrigger.predicateForEvaluatingTrigger(occurringBefore:
dateComponents!)

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 452 ไปถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้


```

case .after:

    predicate =
HMEventTrigger.predicateForEvaluatingTrigger(occurringAfter:
dateComponents!)

case .at:

    predicate =
HMEventTrigger.predicateForEvaluatingTrigger(occurringOn:
dateComponents!)
}

case .sun:

    let significantEventString = (sunState == .sunrise) ?
HMSignificantEvent.sunrise :
HMSignificantEvent.sunset

    switch order {

        case .before:

```

```

        predicate =
HMEventTrigger.predicateForEvaluatingTrigger(occurringBefore:
significantEventString.rawValue, applyingOffset: nil)

        case .after:

            predicate =
HMEventTrigger.predicateForEvaluatingTrigger(occurringAfter:
significantEventString.rawValue, applyingOffset: nil)

            case .at:
                // Significant events must be specified 'before' or 'after'.
                break
            }
        }

        if let predicate = predicate {

            triggerCreator?.addCondition(predicate)

        }

        self.dismiss(animated: true, completion: nil)

```

```

    }

    /// Cancels the creation of the conditions and exits.

    @IBAction func dismiss(_ sender: UIBarButtonItem) {

        self.dismiss(animated: true, completion: nil)

    }

}

HomeListViewController
import UIKit
import HomeKit

/// A generic view controller for displaying a list of homes in a home manager.

class HomeListViewController: HMCatalogViewController,
HMHomeManagerDelegate {

    // MARK: Types

    struct Identifiers {

```

```
static let homeCell = "HomeCell"

static let showHomeSegue = "Show Home"

}
```

```
// MARK: Properties
```

```
var homes = [HMHome]()

var homeManager: HMHomeManager {
    return homeStore.homeManager
}
```

```
// MARK: View Methods
```

```
/// Configures the table view.
```

```
override func viewDidLoad() {
```

```
    super.viewDidLoad()
```

```
tableView.estimatedRowHeight = 44.0
```

```
tableView.rowHeight = UITableView.automaticDimension
```

```
}
```

```
/// Resets the list of homes (which will update the view).
```

```
override func viewWillAppear(_ animated: Bool) {
```

```
    super.viewWillAppear(animated)
```

```
    resetHomesList()
```

```
}
```

```
// MARK: Delegate Registration
```

```
/**
```

```
Registers as the delegate for the home manager and all homes in the internal
```

```
homes list.
```

```
*/
```

```
override func registerAsDelegate() {
```

```
homeManager.delegate = self
```

```
for home in homes {
```

```
    home.delegate = self
```

```
}
```

```
}
```

```
/// Sets the home store's current home based on which cell was selected.
```

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
```

```
    super.prepare(for: segue, sender: sender)
```

```
    if segue.identifier == Identifiers.showHomeSegue {
```

```
        if sender as AnyObject? === self {
```

```
            // Don't update the selected home if we sent ourselves here.
```

```
            return
```

```
        }
```

```
        if let indexPath = tableView.indexPath(for: sender as! UITableViewCell) {
```



```

        homeStore.home = homes[indexPath.row]

    }

}

}

```

// MARK: Table View Methods

/**

Provides the number of sections based on the home array count.

Updates the background message for the table view.

- returns: The number of homes in the internal array.

*/

```

override func tableView(_ tableView: UITableView, numberOfRowsInSectionSection
section: Int) -> Int {

```

```

    let rows = homes.count

```

```

    if rows == 0 {

```

```

let message = NSLocalizedString("No Homes", comment: "No Homes")

setBackgroundMessage(message)

}

else {

setBackgroundMessage(nil)

}

return rows
}

/**
Generates a basic cell for a home.

Subtext is provided to tell the user if the home is shared or owned by the user.

*/

override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) ->
UITableViewCell {

```

```
let cell = tableView.dequeueReusableCell(withIdentifier: Identifiers.homeCell,  
for: indexPath)
```

```
let home = homes[indexPath.row]
```

```
cell.textLabel?.text = home.name
```

```
cell.detailTextLabel?.text = sharedTextForHome(home)
```

```
return cell
```

```
}
```

```
// MARK: Helper Methods
```

```
/**
```

```
Provides an ordering for homes.
```

Homes are first ordered by their 'shared' status, then by name.

- parameter home1: The first HMHome.

- parameter home2: The second HMHome.

- returns: true if home1 is ordered before home2; false otherwise.

*/

```
private func orderHomes(_ home1: HMHome, home2: HMHome) -> Bool {
```

```
    if home1.isAdmin == home2.isAdmin {
```

```
        /*
```

We are comparing two shared homes or two of our homes, just compare names.

```
        */
```

```
        return home1.name.localizedCompare(home2.name) == .orderedAscending
```

```
    }
```

```
    else {
```

```
        /*
```

We are comparing a shared home and one of our homes, if home1 is ours, put it first.

```
        */
```

```
        return home1.isAdmin
```

```
}
```

```
}
```

```
/**
```

Regenerates the list of homes using list provided by the home manager.

The list is then sorted and the view is reloaded.

```
*/
```

```
private func resetHomesList() {
```

```
    homes = homeManager.homes.sorted(by: orderHomes)
```

```
    tableView.reloadData()
```

```
}
```

```
/// Sorts the list of homes (without reloading from the home manager).
```

```
func sortHomes() {
```

```
    homes.sort(by: orderHomes)
```

```
}
```

```
/**
```

Adds a new home into the internal homes array and inserts the new row into the table view.

- parameter home: The new HMHome that's been added.

*/

```
func didAddHome(_ home: HMHome) {  
    homes.append(home)  
    sortHomes()  
    if let newHomeIndex = homes.firstIndex(of: home) {  
        let indexPathOfNewHome = IndexPath(row: newHomeIndex, section: 0)  
        tableView.insertRows(at: [indexPathOfNewHome], with: .automatic)  
    }  
}
```

/**

Removes a home from the internal homes array (if it exists) and deletes corresponding row from the table view.

- parameter home: The HMHome to remove.

*/

```
func didRemoveHome(_ home: HMHome) {  
    guard let removedHomeIndex = homes.firstIndex(of: home) else { return }  
    homes.remove(at: removedHomeIndex)  
    let indexPath = IndexPath(row: removedHomeIndex, section: 0)  
    tableView.deleteRows(at: [indexPath], with: .automatic)  
}
```

/**

- returns: A localized description of who owns the provided home.

- parameter home: The HMHome to describe.

*/

```

func sharedTextForHome(_ home: HMHome) -> String {

    if !home.isAdmin {

        return NSLocalizedString("Shared with Me", comment: "Shared with Me")

    }

    else {

        return NSLocalizedString("My Home", comment: "My Home")

    }

}

// MARK: HMHomeDelegate Methods

/// Finds the cell with corresponds to the provided home and reloads it.

func homeDidUpdateName(_ home: HMHome) {

    if let homeIndex = homes.firstIndex(of: home) {

        let indexPath = IndexPath(row: homeIndex, section: 0)

        tableView.reloadRows(at: [indexPath], with: .automatic)

    }
}

```

```
}
```

```
// MARK: HMHomeManagerDelegate Methods
```

```
/**
```

```
    Reloads data and view.
```

```
    This view controller, in most cases, will remain the home manager delegate.
```

```
    For this reason, this method will close all modal views and pop all detail views
    if the home store's current home is no longer in the home manager's list of
    homes.
```

```
*/
```

```
func homeManagerDidUpdateHomes(_ manager: HMHomeManager) {
```

```
    registerAsDelegate()
```

```
    resetHomesList()
```

```
    if let home = homeStore.home , !manager.homes.contains(home) {
```

```
        // Close all modal and detail views.
```

```

dismiss(animated: true, completion: nil)

_ = navigationController?.popToRootViewController(animated: true)

}

}

/// Registers for the delegate of the new home and updates the view.

func homeManager(_ manager: HMHomeManager, didAdd home: HMHome) {

    home.delegate = self

    didAddHome(home)

}

/**

```

Removes the home from the current list of homes and updates the view.

If the removed home was the current home, this view controller will dismiss

all modals views and pop all detail views.

*/

```

func homeManager(_ manager: HMHomeManager, didRemove home: HMHome)
{

    didRemoveHome(home)


    guard let selectedHome = homeStore.home , home == selectedHome else {
return }


    homeStore.home = nil

    // Close all modal and detail views.
    dismiss(animated: true, completion: nil)

    _ = navigationController?.popToRootViewController(animated: true)

    }
}

```

HomeController

```
import Foundation
```

```
import UIKit
```

```
import HomeKit
```

/// Distinguishes between the three types of cells in the HomeViewController.

```
enum HomeCellType {
```

/// Represents an actual object in HomeKit.

```
case object
```

/// Represents an "Add" row for users to select to create an object in HomeKit.

```
case add
```

/// The cell is displaying text to show the user that no objects exist in this section.

```
case none
```

```
}
```

```
/**
```

A view controller that displays all elements within a home.

It contains separate sections for Accessories, Rooms, Zones, Action Sets,

Triggers, and Service Groups.

```
*/
```



```

class HomeViewController: HMCatalogViewController, HMAccessoryDelegate {

    // MARK: Types

    struct Identifiers {

        static let addCell = "AddCell"

        static let disabledAddCell = "DisabledAddCell"

        // static let accessoryCell = "AccessoryCell"
        // static let unreachableAccessoryCell = "UnreachableAccessoryCell"
        // static let roomCell = "RoomCell"
        //static let zoneCell = "ZoneCell"
        // static let userCell = "UserCell"

        static let actionSetCell = "ActionSetCell"

        static let triggerCell = "TriggerCell"

        // static let serviceGroupCell = "ServiceGroupCell"

        static let addTimerTriggerSegue = "Add Timer Trigger"

        static let addCharacteristicTriggerSegue = "Add Characteristic Trigger"

        static let addLocationTriggerSegue = "Add Location Trigger"

        static let addActionSetSegue = "Add Action Set"
    }
}

```

เอกสารนี้เป็นเอกสารที่สงวนลิขสิทธิ์สำหรับการใช้ในเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า

ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 471 ไปถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

static let addAccessoriesSegue = "Add Accessories"

// static let showRoomSegue = "Show Room"

//static let showZoneSegue = "Show Zone"

static let showActionSetSegue = "Show Action Set"

static let showServiceGroupSegue = "Show Service Group"

// static let showAccessorySegue = "Show Accessory"

static let modifyAccessorySegue = "Modify Accessory"

static let showTimerTriggerSegue = "Show Timer Trigger"

static let showLocationTriggerSegue = "Show Location Trigger"

static let showCharacteristicTriggerSegue = "Show Characteristic Trigger"

}

// MARK: Properties

/// A structure to maintain internal arrays of HomeKit objects.

private var objectCollection = HomeKitObjectCollection()

// MARK: View Methods

```

```
/**
```

Determines the destination of the segue and passes the correct

HomeKit object onto the next view controller.

```
*/
```

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
```

```
    guard let sender = sender as? UITableViewCell else { return }
```

```
    guard let indexPath = tableView.indexPath(for: sender) else { return }
```

```
    let homeKitObject = homeKitObjectAtIndexPath(indexPath)
```

```
    let destinationViewController = segue.intendedDestinationViewController
```

```
    switch segue.identifier! {
```

```
//        case Identifiers.showRoomSegue:
```

```
//            let roomVC = destinationViewController as! RoomViewController
```

```
//            roomVC.room = homeKitObject as? HMRoom
```

```
//        case Identifiers.showZoneSegue:
```

เอกสารนี้เป็นเอกสารทูลสงวนลิขสิทธิ์การสงวนเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า

ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 473 จึงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

//          let zoneViewController = destinationViewController as!
ZoneViewController

//          zoneViewController.homeZone = homeKitObject as? HMZone


case Identifiers.showActionSetSegue:

    let          actionSetVC          =          destinationViewController          as!
ActionSetViewController

    actionSetVC.actionSet = homeKitObject as? HMActionSet


case Identifiers.showServiceGroupSegue:

    let          serviceGroupVC          =          destinationViewController          as!
ServiceGroupViewController

    serviceGroupVC.serviceGroup = homeKitObject as? HMServiceGroup


//          case Identifiers.showAccessorySegue:


//          let detailVC = destinationViewController as! ServicesViewController

/*

The services view controller is generic, we need to provide

showsFavorites to display the stars next to characteristics.

```

```

        */

//      detailVC.accessory = homeKitObject as? HMAccessory

//      detailVC.showsFavorites = true

//      detailVC.cellDelegate = AccessoryUpdateController()

        case Identifiers.modifyAccessorySegue:

            let      addAccessoryVC      =      destinationViewController      as!
ModifyAccessoryViewController

            addAccessoryVC.accessory = homeKitObject as? HMAccessory

        case Identifiers.showTimerTriggerSegue:

            let      triggerVC      =      destinationViewController      as!
TimerTriggerViewController

            triggerVC.trigger = homeKitObject as? HMTimerTrigger

        case Identifiers.showLocationTriggerSegue:

            let      triggerVC      =      destinationViewController      as!
LocationTriggerViewController

            triggerVC.trigger = homeKitObject as? HMEventTrigger

```

```

case Identifiers.showCharacteristicTriggerSegue:

    let triggerVC = destinationViewController as!
    CharacteristicTriggerViewController

    triggerVC.trigger = homeKitObject as? HMEventTrigger

default:

    print("Received unknown segue identifier: \(segue.identifier ?? "nil").")
}
}

/// Configures the table view.

override func awakeFromNib() {

    super.awakeFromNib()

    tableView.estimatedRowHeight = 44.0

    tableView.rowHeight = UITableView.automaticDimension

}

```



```

/// Sets the navigation title and reloads view.

override func viewWillAppear(_ animated: Bool) {

    super.viewWillAppear(animated)

    self.navigationItem.setHidesBackButton(true, animated: true);

    navigationItem.title = home.name

    reloadData()
}

// MARK: Delegate Registration

/**

Registers as the delegate for the home store's current home

and all accessories in the home.

*/

override func registerAsDelegate() {

    super.registerAsDelegate()

```

```
for accessory in home.accessories {  
  
    accessory.delegate = self  
  
}  
  
}  
  
  
// MARK: Helper Methods  
  
/// Resets the object collection and reloads the view.  
private func reloadData() {  
    objectCollection.resetWithHome(home)  
    tableView.reloadData()  
}  
  
/**  
  
    Determines the type of the cell based on the index path.  
  
    - parameter indexPath: The NSIndexPath of the cell.
```

- returns: The HomeCellType for cell.

*/

```
private func cellTypeForIndexPath(_ indexPath: IndexPath) -> HomeCellType {  
  
    guard let section = HomeKitObjectSection(rawValue: indexPath.section) else {  
return .none }  
  
    let objectCount = objectCollection.objectsForSection(section).count  
  
    if objectCount == 0 {  
        // No objects -- this is either an 'Add Row' or a 'None Row'.  
        return home.isAdmin ? .add : .none  
    }  
  
    else if indexPath.row == objectCount {  
  
        return .add  
  
    }  
  
    else {  
  
        return .object  
  
    }  
}
```

```

    }

    /// Reloads the trigger section.

    private func updateTriggerAddRow() {

        let triggerSection = IndexSet(integer:
HomeKitObjectSection.trigger.rawValue)

        tableView.reloadSections(triggerSection, with: .automatic)
    }

    /// Reloads the action set section.

    private func updateActionSetSection() {

        let actionSetSection = IndexSet(integer:
HomeKitObjectSection.actionSet.rawValue)

        tableView.reloadSections(actionSetSection, with: .automatic)

        updateTriggerAddRow()
    }

```

```

    }

    /// - returns: true if there are accessories within the home; false otherwise.

    // private var canAddActionSet: Bool {

    //     return !objectCollection.accessories.isEmpty

    // }

    /// - returns: true if there are action sets (with actions) within the home; false
    otherwise.

    private var canAddTrigger: Bool {

        return objectCollection.actionSets.contains { actionSet in

            return !actionSet.actions.isEmpty

        }

    }

}

/**

Provides the 'HomeKit object' (AnyObject?) at the specified index path.

```

- parameter indexPath: The NSIndexPath of the object.

- returns: The HomeKit object.

*/

```
private func homeKitObjectAtIndexPath(_ indexPath: IndexPath) -> AnyObject?
{
    if cellTypeForIndexPath(indexPath) != .object {
        return nil
    }

    if let section = HomeKitObjectSection(rawValue: indexPath.section) {
        return objectCollection.objectsForSection(section)[indexPath.row]
    }

    return nil
}
```

// MARK: Table View Methods


```

/// - returns: The number of HomeKitObjectSections.

override func numberOfSections(in tableView: UITableView) -> Int {

    return HomeKitObjectSection.count

}

/// - returns: Localized titles for each of the HomeKit sections.

override func tableView(_ tableView: UITableView, titleForHeaderInSection
section: Int) ->
String? {

    switch HomeKitObjectSection(rawValue: section) {

//        case .accessory?:

//            return NSLocalizedString("Accessories", comment: "Accessories")

//        case .room?:

//            return NSLocalizedString("Rooms", comment: "Rooms")

//        case .zone?:

```

```

//      return NSLocalizedString("Zones", comment: "Zones")

//      case .user?:

//      return NSLocalizedString("Users", comment: "Users")

//

case .actionSet?:

    return NSLocalizedString("Scenes", comment: "Scenes")

case .trigger?:

    return NSLocalizedString("Triggers", comment: "Triggers")

//      case .serviceGroup?:

//      return NSLocalizedString("Service Groups", comment: "Service Groups")

//

case nil:

    fatalError("Unexpected HomeKitObjectSection raw value.")

}

```

```

}

/// - returns: Localized text for the 'add row'.

private func titleForAddRowInSection(_ section: HomeKitObjectSection) ->
String {

    switch section {

//      case .accessory:

//          return NSLocalizedString("Add Accessory...", comment: "Add
Accessory")

//      case .room:

//          return NSLocalizedString("Add Room...", comment: "Add Room")

//      case .zone:

//          return NSLocalizedString("Add Zone...", comment: "Add Zone")

//      case .user:

//          return NSLocalizedString("Manage Users...", comment: "Manage Users")

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 485 ังอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

case .actionSet:

    return NSLocalizedString("Add Scene...", comment: "Add Scene")

case .trigger:

    return NSLocalizedString("Add Trigger...", comment: "Add Trigger")

// case .serviceGroup:
//     return NSLocalizedString("Add Service Group...", comment: "Add
Service Group")
}
}

/// - returns: Localized text for the 'none row'.

private func titleForNoneRowInSection(_ section: HomeKitObjectSection) ->
String {

    switch section {

// case .accessory:

```

```

//          return NSLocalizedString("No Accessories...", comment: "No
Accessories")

//      case .room:

//          return NSLocalizedString("No Rooms...", comment: "No Rooms")

//      case .zone:

//          return NSLocalizedString("No Zones...", comment: "No Zones")

//      case .user:

//          // We only ever list 'Manage Users'.

//          return NSLocalizedString("Manage Users...", comment: "Manage Users")

      case .actionSet:

          return NSLocalizedString("No Scenes...", comment: "No Scenes")

      case .trigger:

          return NSLocalizedString("No Triggers...", comment: "No Triggers")

```

```

//      case .serviceGroup:

//          return NSLocalizedString("No Service Groups...", comment: "No Service
Groups")

        }

    }

    /// - returns: Localized descriptions for HomeKit object types.
    override func tableView(_ tableView: UITableView, titleForFooterInSection
section: Int) ->
String? {

        switch HomeKitObjectSection(rawValue: section) {

//            case .zone?:

//                return NSLocalizedString("Zones are optional collections of rooms.",
comment: "Zones

Description")

//            case .user?:

```



```
//          return NSLocalizedString("Users can control the accessories in your
home. You can

share your home with anybody with an iCloud account.", comment: "Users
Description")

case .actionSet?:

    return NSLocalizedString("Scenes (action sets) represent a state of your
home. You must

have at least one paired accessory to create a scene.", comment: "Scenes
Description")

case .trigger?:

    return NSLocalizedString("Triggers set scenes at specific times, when you
get to locations,

or when a characteristic is in a specific state. You must have created at least one scene
with an

action to create a trigger.", comment: "Trigger Description")

//          case .serviceGroup?:

//          return NSLocalizedString("Service groups organize services in a custom
way. For
```

example, add a subset of lights in your living room to control them without controlling all the

lights in the living room.", comment: "Service Group Description")

case nil:

fatalError("Unexpected HomeKitObjectSection raw value.")

default:

return nil

}

}

/**

Provides the number of rows in each HomeKit object section.

Most sections just return the object count, but we also handle special cases.

*/

override func tableView(_ tableView: UITableView, numberOfRowsInSectionSection section: Int) -> Int {

```

let sectionEnum = HomeKitObjectSection(rawValue: section)!

// Only "Manage Users" button is in the Users section

// if sectionEnum == .user {

//     return 1

// }

let objectCount = objectCollection.objectsForSection(sectionEnum).count
if home.isAdmin {
    // For add row.
    return objectCount + 1
}

else {

    // Always show at least one row in the section.

    return max(objectCount, 1)

}

}

```

```

/// Generates a cell based on it's computed type.

override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) ->
UITableViewCell {

    switch cellTypeForIndexPath(indexPath) {

    case .add:

        return self.tableView(tableView, addCellForRowAtIndexPath: indexPath)

    case .object:

        return self.tableView(tableView,
homeKitObjectCellForRowAtIndexPath: indexPath)

    case .none:

        return self.tableView(tableView, noneCellForRowAtIndexPath:
indexPath)

    }

}

}

/// Generates a 'none cell' with a localized title.

```

```
private func tableView(_ tableView: UITableView, noneCellForRowAtIndexPath  
indexPath:
```

```
IndexPath) -> UITableViewCell {
```

```
    let cell = tableView.dequeueReusableCell(withIdentifier:  
Identifiers.disabledAddCell, for:
```

```
indexPath)
```

```
    let section = HomeKitObjectSection(rawValue: indexPath.section)!
```

```
    cell.textLabel!.text = titleForRowInSection(section)
```

```
    return cell
```

```
}
```

```
/**
```

Generates an 'add cell' with a localized title.

In some cases, the 'add cell' will be 'disabled' because the user is not

allowed to perform the action.

*/

private func tableView(_ tableView: UITableView, addCellForRowAtIndexPath
indexPath:

IndexPath) -> UITableViewCell {

var reuseIdentifier = Identifiers.addCell

let section = HomeKitObjectSection(rawValue: indexPath.section)

// if (!canAddActionSet && section == .actionSet) ||

// (!canAddTrigger && section == .trigger) || !home.isAdmin {

// reuseIdentifier = Identifiers.disabledAddCell

// }

let cell = tableView.dequeueReusableCell(withIdentifier: reuseIdentifier, for:
indexPath)

cell.textLabel!.text = titleForAddRowInSection(section!)


```
return cell
```

```
}
```

```
/**
```

Produces the cell reuse identifier based on the section.

- parameter indexPath: The NSIndexPath of the cell.

- returns: The cell reuse identifier.

```
*/
```

```
private func reuseIdentifierForIndexPath(_ indexPath: IndexPath) -> String {
```

```
    switch HomeKitObjectSection(rawValue: indexPath.section) {
```

```
//        case .accessory?:
```

```
//            let accessory = homeKitObjectAtIndexPath(indexPath) as! HMAccessory
```

```
//            return accessory.isReachable ? Identifiers.accessoryCell :
```

```
Identifiers.unreachableAccessoryCell
```

```
//
```

```

//      case .room?:

//          return Identifiers.roomCell


//      case .zone?:

//          return Identifiers.zoneCell


//      case .user?:

//          return Identifiers.userCell

      case .actionSet?:

          return Identifiers.actionSetCell


      case .trigger?:

          return Identifiers.triggerCell


//      case .serviceGroup?:

//          return Identifiers.serviceGroupCell

```

```

        case nil:

            fatalError("Unexpected HomeKitObjectSection raw value.")

        }

    }

    /// Generates a cell for the HomeKit object at the specified index path.

    private func tableView(_ tableView: UITableView,
homeKitObjectCellForRowAtIndexPath
indexPath: IndexPath) -> UITableViewCell {

        // Grab the object associated with this indexPath.

        let homeKitObject = homeKitObjectAtIndex(indexPath)

        // Get the name of the object.

        let name: String

        switch HomeKitObjectSection(rawValue: indexPath.section) {

            // case .accessory?:

            //     let accessory = homeKitObject as! HMAccessory

            //     name = accessory.name

```

```

//      case .room?:

//          let room = homeKitObject as! HMRoom

//          name = self.home.nameForRoom(room)


//      case .zone?:

//          let zone = homeKitObject as! HMZone

//          name = zone.name

//      case .user?:

//          name = ""

case .actionSet?:

    let actionSet = homeKitObject as! HMActionSet

    name = actionSet.name


case .trigger?:

    let trigger = homeKitObject as! HMTrigger

    name = trigger.name

```

```

//      case .serviceGroup?:

//          let serviceGroup = homeKitObject as! HMServiceGroup

//          name = serviceGroup.name


    case nil:

        fatalError("Unexpected HomeKitObjectSection raw value.")
    }

// Grab the appropriate reuse identifier for this index path.
let reuseIdentifier = reuseIdentifierForIndexPath(indexPath)

let cell = tableView.dequeueReusableCell(withIdentifier: reuseIdentifier, for:
indexPath)

cell.textLabel?.text = name


return cell

}

```

```

/// Allows users to remove HomeKit object rows if they are the admin of the home.

override func tableView(_ tableView: UITableView, canEditRowAt indexPath:
IndexPath) -> Bool
{

    let homeKitObject = homeKitObjectAtIndex(indexPath)

    if !home.isAdmin {
        return false
    }

    if let actionSet = homeKitObject as? HMActionSet, actionSet.isBuiltIn {
        // We cannot remove built-in action sets.
        return false
    }

    // Any row that is not an 'add' row, and is not the roomForEntireHome, can be
    removed.

    return !(homeKitObject as? NSObject == home.roomForEntireHome) ||

```



```

cellTypeForIndexPath(indexPath) == .add)

}

/// Removes the HomeKit object at the specified index path.

override func tableView(_ tableView: UITableView, commit editingStyle:
UITableViewCell.EditingStyle, forRowAt indexPath: IndexPath) {

    if editingStyle == .delete {

        let homeKitObject = homeKitObjectAtIndexPath(indexPath)!

        // Remove the object from the data structure. If it fails put it back.
        didRemoveHomeKitObject(homeKitObject)

        removeHomeKitObject(homeKitObject) { error in

            guard let error = error else { return }

            self.displayError(error)

            self.didAddHomeKitObject(homeKitObject)

        }

    }

}

```

```

    }

    /// Handles cell selection based on the cell type.

    override func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {

        tableView.deselectRow(at: indexPath, animated: true)

        let cell = tableView.cellForRow(at: indexPath)!

        guard cell.selectionStyle != .none else { return }

        guard let section = HomeKitObjectSection(rawValue: indexPath.section) else {
            fatalError("Unexpected HomeKitObjectSection raw value.")
        }

        if cellTypeForIndexPath(indexPath) == .add{

            switch section {

                // case .accessory:

                // browseForAccessories()

```

```
// case .room:

//     addNewRoom()

// case .zone:

//     addNewZone()

// case .user:

//     manageUsers()

case .actionSet:

    addNewActionSet()

case .trigger:

    self.performSegue(withIdentifier: Identifiers.addTimerTriggerSegue,
sender: self)

// case .serviceGroup:
```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 503 จึงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

//      addNewServiceGroup()

    }

}

else if section == .actionSet {

    let    selectedActionSet    =    homeKitObjectAtIndexPath(indexPath)    as!
HMActionSet

    executeActionSet(selectedActionSet)

}

}

/// Handles an accessory button tap based on the section.

override    func    tableView(_    tableView:    UITableView,
accessoryButtonTappedForRowWith

indexPath: IndexPath) {

    let cell = tableView.cellForRow(at: indexPath)

    if HomeKitObjectSection(rawValue: indexPath.section) == .trigger {

        let trigger = homeKitObjectAtIndexPath(indexPath)

```

```

switch trigger {

    case is HMTimerTrigger:

        performSegue(withIdentifier: Identifiers.showTimerTriggerSegue,
sender: cell)

    case let eventTrigger as HMEventTrigger:

        if eventTrigger.isLocationEvent {

            performSegue(withIdentifier:
Identifiers.showLocationTriggerSegue, sender: cell)

        }

        else {

            performSegue(withIdentifier:
Identifiers.showCharacteristicTriggerSegue, sender:
cell)

        }

    default: break

}

```

```

    }

}

// MARK: Action Methods

/// Presents an alert controller to allow the user to choose a trigger type.

// private func addNewTrigger() {

//     let title = NSLocalizedString("Add Trigger", comment: "Add Trigger")

//     let alertController = UIAlertController(title: title, message: nil, preferredStyle:
.actionSheet)

//

//     // Timer trigger

//     let timeAction = UIAlertAction(title: NSLocalizedString("Time", comment:
"Time"), style:

.default) { _ in

//         self.performSegue(withIdentifier: Identifiers.addTimerTriggerSegue, sender:
self)

//     }

//     alertController.addAction(timeAction)

```



```

//

// // Characteristic trigger

//

//

// // Location trigger

//

//

// // Cancel

//     let cancelAction = UIAlertAction(title: NSLocalizedString("Cancel",
comment: "Cancel"),
style: .cancel, handler: nil)

//     alertController.addAction(cancelAction)

//

// // Present alert

//     if let presenter = alertController.popoverPresentationController {

//

//         presenter.sourceView = self.view

//         presenter.sourceRect = self.view.frame

```

```

// presenter.permittedArrowDirections = UIPopoverArrowDirection(rawValue:
0)

// }

// self.present(alertController, animated: true, completion: nil)

// }

/// Navigates into the action set view controller.

private func addNewActionSet() {

performSegue(withIdentifier: Identifiers.addActionSetSegue, sender: self)

}

/// Navigates into the browse accessory view controller.

private func browseForAccessories() {

performSegue(withIdentifier: Identifiers.addAccessoriesSegue, sender: self)

}

// MARK: Dialog Creation Methods

```

```
/// Presents a dialog to name a new room and generates the HomeKit object if
committed.
```

```
private func addNewRoom() {
```

```
    presentAddAlertWithType(NSLocalizedString("Room", comment:
"Room"),
```

```
    placeholder: NSLocalizedString("Living Room", comment: "Living Room"))
{ roomName in
```

```
    self.addRoomWithName(roomName)
```

```
}
```

```
}
```

```
/// Presents a dialog to name a new service group and generates the HomeKit object
if
committed.
```

```
private func addNewServiceGroup() {
```

```
    presentAddAlertWithType(NSLocalizedString("Service      Group",
comment: "Service
```

```
Group"),
```

```
    placeholder:  NSLocalizedString("Group",  comment:  "Group"))  {
groupName in
```

```

self.addServiceGroupWithName(groupName)

}

}

/// Presents a dialog to name a new zone and generates the HomeKit object if
committed.

private func addNewZone() {

    presentAddAlertWithAttributeType(NSLocalizedString("Zone", comment:
"Zone"),
        placeholder: NSLocalizedString("Upstairs", comment: "Upstairs")) {
zoneName in
        self.addZoneWithName(zoneName)
    }
}

}

// MARK: HomeKit Object Creation and Deletion

/**

Switches based on the type of object attempts to remove the HomeKit object

```

from the current home.

- parameter object: The HomeKit object to remove.

- parameter completionHandler: The closure to invoke when the removal has been

completed.

```
*/
```

```
private func removeHomeKitObject(_ object: AnyObject, completionHandler:  
@escaping
```

```
(Error?) -> Void) {
```

```
    switch object {
```

```
        case let actionSet as HMAActionSet:
```

```
            home.removeActionSet(actionSet) { error in
```

```
                completionHandler(error)
```

```
                self.updateActionSetSection()
```

```
            }
```

```
        case let accessory as HMAAccessory:
```

```
home.removeAccessory(accessory, completionHandler: completionHandler)
```

case let room as HMRoom:

```
home.removeRoom(room, completionHandler: completionHandler)
```

case let zone as HMZone:

```
home.removeZone(zone, completionHandler: completionHandler)
```

case let trigger as HMTrigger:

```
home.removeTrigger(trigger, completionHandler: completionHandler)
```

case let serviceGroup as HMServiceGroup:

```
home.removeServiceGroup(serviceGroup, completionHandler: completionHandler)
```

default:

```
fatalError("Attempted to remove unknown HomeKit object.")
```



```

    }

}

/**

Adds a room to the current home.

- parameter name: The name of the new room.

*/
private func addRoomWithName(_ name: String) {
    home.addRoom(withName: name) { newRoom, error in
        if let error = error {
            self.displayError(error)
        }
        return
    }

    self.didAddHomeKitObject(newRoom)
}
}

```

/**

Adds a service group to the current home.

- parameter name: The name of the new service group.

*/

```
private func addServiceGroupWithName(_ name: String) {  
    home.addServiceGroup(withName: name) { newGroup, error in  
        if let error = error {  
            self.displayError(error)  
            return  
        }  
        self.didAddHomeKitObject(newGroup)  
    }  
}
```

/**

Adds a zone to the current home.

- parameter name: The name of the new zone.

*/

```
private func addZoneWithName(_ name: String) {  
  
    home.addZone(withName: name) { newZone, error in  
  
        if let error = error {  
            self.displayError(error)  
            return  
        }  
  
        self.didAddHomeKitObject(newZone)  
  
    }  
  
}
```

/// Presents modal view for managing users.

```
private func manageUsers() {
```

```
    home.manageUsers { error in
```

```

        if let error = error {

            self.displayError(error)

        }

    }

}

```

```

/**

```

Checks to see if an action set has any actions.

If actions exists, the action set will be executed.

Otherwise, the user will be alerted.

- parameter actionSet: The HMActionSet to evaluate and execute.

```

*/

```

```

private func executeActionSet(_ actionSet: HMActionSet) {

```

```

    if actionSet.actions.isEmpty {

```

```

        let alertTitle = NSLocalizedString("Empty Scene", comment: "Empty
Scene")

```

```
let alertMessage = NSLocalizedString("This scene is empty. To set this  
scene, first add some
```

```
actions to it.", comment: "Empty Scene Description")
```

```
displayMessage(alertTitle, message: alertMessage)
```

```
return
```

```
}
```

```
home.executeActionSet(actionSet) { error in
```

```
guard let error = error else { return }
```

```
self.displayError(error)
```

```
}
```

```
}
```

```
/**
```

Adds the HomeKit object into the object collection and inserts the new row into the section.

- parameter object: The HomeKit object to add.

*/

```
private func didAddHomeKitObject(_ object: AnyObject?) {  
  
    if let object = object {  
  
        objectCollection.append(object)  
  
        if let newIndexPath = objectCollection.indexPathOfObject(object) {  
            tableView.insertRows(at: [newIndexPath], with: .automatic)  
        }  
    }  
}
```

/**

Finds the NSIndexPath of the specified object and reloads it in the table view.

- parameter object: The HomeKit object that was modified.

*/

```
private func didModifyHomeKitObject(_ object: AnyObject?) {
```

```
    if let object = object,
```

```
        let indexPath = objectCollection.indexPathOfObject(object) {
```

```
            tableView.reloadRows(at: [indexPath], with: .automatic)
```

```
        }
```

```
    }
```

/**

Removes the HomeKit object from the object collection and then deletes the row from the section.

- parameter object: The HomeKit object to remove.

*/

```
private func didRemoveHomeKitObject(_ object: AnyObject?) {
```

```
    if let object = object,
```

```
        let indexPath = objectCollection.indexPathOfObject(object) {
```

```

        objectCollection.remove(object)

        tableView.deleteRows(at: [objectIndexPath], with: .automatic)

    }

}

```

```

/*

```

The following methods call the above helper methods to handle the addition, removal, and modification of HomeKit objects.

```

*/

```

```

// MARK: HMHomeDelegate Methods

```

```

func homeDidUpdateName(_ home: HMHome) {

    navigationItem.title = home.name

    reloadData()

}

```

```

func home(_ home: HMHome, didAddAccessory accessory: HMAccessory) {

```

```

didAddHomeKitObject(accessory)

accessory.delegate = self

}

func home(_ home: HMHome, didRemoveAccessory accessory: HMAccessory) {

    didRemoveHomeKitObject(accessory)

}

// MARK: Triggers

func home(_ home: HMHome, didAddTrigger trigger: HMTrigger) {

    didAddHomeKitObject(trigger)

}

func home(_ home: HMHome, didRemoveTrigger trigger: HMTrigger) {

    didRemoveHomeKitObject(trigger)

}

```

```

func home(_ home: HMHome, didUpdateNameFor trigger: HMTrigger) {

    didModifyHomeKitObject(trigger)

}

// MARK: Service Groups

func home(_ home: HMHome, didAddServiceGroup group: HMServiceGroup) {

    didAddHomeKitObject(group)

}

func home(_ home: HMHome, didRemoveServiceGroup group:
HMServiceGroup) {

    didRemoveHomeKitObject(group)

}

@objc(home:didUpdateNameForGroup:)

func home(_ home: HMHome, didUpdateNameFor group: HMServiceGroup) {

    didModifyHomeKitObject(group)

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 522 ไปถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```
}
```

```
// MARK: Action Sets
```

```
func home(_ home: HMHome, didAddActionSet actionSet: HMActionSet) {
```

```
    didAddHomeKitObject(actionSet)
```

```
}
```

```
func home(_ home: HMHome, didRemoveActionSet actionSet: HMActionSet) {
```

```
    didRemoveHomeKitObject(actionSet)
```

```
}
```

```
@objc(home:didUpdateNameForActionSet:)
```

```
func home(_ home: HMHome, didUpdateNameFor actionSet: HMActionSet) {
```

```
    didModifyHomeKitObject(actionSet)
```

```
}
```

```
// MARK: Zones
```

```
func home(_ home: HMHome, didAddZone zone: HMZone) {
```

```
    didAddHomeKitObject(zone)
```

```
}
```

```
func home(_ home: HMHome, didRemoveZone zone: HMZone) {
```

```
    didRemoveHomeKitObject(zone)
```

```
}
```

```
@objc(home;didUpdateNameForZone:)
```

```
func home(_ home: HMHome, didUpdateNameFor zone: HMZone) {
```

```
    didModifyHomeKitObject(zone)
```

```
}
```

```
// MARK: Rooms
```

```
func home(_ home: HMHome, didAddRoom room: HMRoom) {
```

```
    didAddHomeKitObject(room)
```



```
}
```

```
func home(_ home: HMHome, didRemoveRoom room: HMRoom) {
```

```
    didRemoveHomeKitObject(room)
```

```
}
```

```
@objc(home:didUpdateNameForRoom:)
```

```
func home(_ home: HMHome, didUpdateNameFor room: HMRoom) {
```

```
    didModifyHomeKitObject(room)
```

```
}
```

```
// MARK: Accessories
```

```
func accessoryDidUpdateReachability(_ accessory: HMAccessory) {
```

```
    didModifyHomeKitObject(accessory)
```

```
}
```

```
func accessoryDidUpdateName(_ accessory: HMAccessory) {
```

```

        didModifyHomeKitObject(accessory)

    }

}

UIStoryboardSegue+InterndedDestination

import UIKit

extension UIStoryboardSegue {

    /// - returns: The intended UIViewController from the segue's destination.
    var intendedDestinationViewController: UIViewController {

        if let navigationController = self.destination as? UINavigationController {
            return navigationController.topViewController!
        }

        return self.destination

    }

}

HMEventtrigger+Convenience

import HomeKit

```

```
import MapKit
```

```
extension HMEventTrigger {
```

```
    /// - returns: The first CLCircularRegion in the location event regions.
```

```
    var firstCircularLocationEventRegion: CLCircularRegion? {
```

```
        for event in self.locationEvents {
```

```
            if let circularRegion = event.region as? CLCircularRegion {
```

```
                return circularRegion
```

```
            }
```

```
        }
```

```
        return nil
```

```
    }
```

```
    /// - returns: All events in the trigger which are HMLocationEvents.
```

```
    var locationEvents: [HMLocationEvent] {
```

```
        return events.filter { $0 is HMLocationEvent } as! [HMLocationEvent]
```

```
    }
```

/// - returns: true if the trigger contains a location event, false otherwise.

```
var isLocationEvent: Bool {
```

```
  for event in events {
```

```
    if event is HMLocationEvent {
```

```
      return true
```

```
    }
```

```
  }
```

```
  return false
```

```
}
```

```
/**
```

- returns: The first HMEvent in the event list that is

an HMLocationEvent, nil otherwise.

```
*/
```

```
var locationEvent: HMLocationEvent? {
```

```
  for event in events {
```

```
    if let locationEvent = event as? HMLocationEvent {
```

```

        return locationEvent

    }

}

return nil

}

/**
 - returns: All HMEvents in the events list that are
           HMCharacteristicEvents.
 */
var characteristicEvents: [HMCharacteristicEvent<NSCopying>] {

    return events.filter { $0 is HMCharacteristicEvent<NSCopying> } as!

    [HMCharacteristicEvent<NSCopying>]

}

}

```

UITableViewController+Convenience

```
import HomeKit
```

```
import UIKit
```

```
extension UITableViewController {
```

```
/**
```

Displays or hides a label in the background of the table view.

- parameter message: The String message to display. The message is hidden if nil is provided.

```
*/
```

```
func setBackgroundMessage(_ message: String?) {
```

```
    if let message = message {
```

```
        // Display a message when the table is empty
```

```
        let messageLabel = UILabel()
```

```
        messageLabel.text = message
```

```
        messageLabel.font = .preferredFont(forTextStyle: UIFont.TextStyle.body)
```

```
        messageLabel.textColor = .lightGray
```

```
        messageLabel.textAlignment = .center
```



```

messageLabel.sizeToFit()

tableView.backgroundColor = messageLabel

tableView.separatorStyle = .none

}

else {

tableView.backgroundColor = nil
tableView.separatorStyle = .singleLine
}

}

}

NSPredicate+Condition

import HomeKit

/// Represents condition type in HomeKit with associated values.

enum HomeKitConditionType {

/**

Represents a characteristic condition.

```

The tuple represents the HMCharacteristic and its condition value.

For example, "Current gargage door is set to 'Open'".

*/

case characteristic(HMCharacteristic, NSCopying)

/**

Represents a time condition.

The tuple represents the time ordering and the sun state.

For example, "Before sunset".

*/

case sunTime(TimeConditionOrder, TimeConditionSunState)

/**

Represents an exact time condition.

The tuple represents the time ordering and time.

For example, "At 12:00pm".

*/

case exactTime(TimeConditionOrder, DateComponents)

/// The predicate is not a HomeKit condition.

case unknown

}

extension NSPredicate {

/**

Parses the predicate and attempts to generate a characteristic-value
HomeKitConditionType.

- returns: An optional characteristic-value tuple.

*/

private func characteristic() -> HomeKitConditionType? {

guard let predicate = self as? NSCompoundPredicate else { return nil }

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 533 จึงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```
guard let subpredicates = predicate.subpredicates as? [NSPredicate] else {  
return nil }
```

```
guard subpredicates.count == 2 else { return nil }
```

```
var characteristicPredicate: NSComparisonPredicate? = nil
```

```
var valuePredicate: NSComparisonPredicate? = nil
```

```
for subpredicate in subpredicates {
```

```
    if let comparison = subpredicate as? NSComparisonPredicate ,  
comparison.leftExpression.expressionType == .keyPath &&  
comparison.rightExpression.expressionType == .constantValue {
```

```
        switch comparison.leftExpression.keyPath {
```

```
            case HMCharacteristicKeyPath:
```

```
                characteristicPredicate = comparison
```

```
            case HMCharacteristicValueKeyPath:
```

```
                valuePredicate = comparison
```

```

        default:

            break

        }

    }

}

if let characteristic = characteristicPredicate?.rightExpression.constantValue
as?
HMCharacteristic,
let characteristicValue = valuePredicate?.rightExpression.constantValue as?
NSCopying {
    return .characteristic(characteristic, characteristicValue)
}

return nil
}

/**

    Parses the predicate and attempts to generate an order-sunstate
    HomeKitConditionType.

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 535 ังอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

- returns: An optional order-sunstate tuple.

*/

```
private func sunState() -> HomeKitConditionType? {

    guard let comparison = self as? NSComparisonPredicate else { return nil }

    guard comparison.leftExpression.expressionType == .keyPath else { return nil }
}

guard comparison.rightExpression.expressionType == .function else { return nil }
}

guard comparison.rightExpression.function == "now" else { return nil }
guard comparison.rightExpression.arguments?.count == 0 else { return nil }

switch (comparison.leftExpression.keyPath,
comparison.predicateOperatorType) {

    case (HMSignificantEvent.sunrise.rawValue, .lessThan):

        return .sunTime(.after, .sunrise)

    case (HMSignificantEvent.sunrise.rawValue, .lessThanOrEqualTo):

        return .sunTime(.after, .sunrise)
```


case (HMSignificantEvent.sunrise.rawValue, .greaterThan):

return .sunTime(.before, .sunrise)

case (HMSignificantEvent.sunrise.rawValue, .greaterThanOrEqualTo):

return .sunTime(.before, .sunrise)

case (HMSignificantEvent.sunset.rawValue, .lessThan):

return .sunTime(.after, .sunset)

case (HMSignificantEvent.sunset.rawValue, .lessThanOrEqualTo):

return .sunTime(.after, .sunset)

case (HMSignificantEvent.sunset.rawValue, .greaterThan):

return .sunTime(.before, .sunset)

case (HMSignificantEvent.sunset.rawValue, .greaterThanOrEqualTo):

return .sunTime(.before, .sunset)

```

default:

    return nil

}

}

/**
    Parses the predicate and attempts to generate an order-exacttime
    HomeKitConditionType.
    - returns: An optional order-exacttime tuple.
 */

private func exactTime() -> HomeKitConditionType? {

    guard let comparison = self as? NSComparisonPredicate else { return nil }

    guard comparison.leftExpression.expressionType == .function else { return nil }

    guard comparison.leftExpression.function == "now" else { return nil }

    guard comparison.rightExpression.expressionType == .constantValue else {
return nil }

```

```

guard let dateComponents = comparison.rightExpression.constantValue as?
DateComponents

else { return nil }

switch comparison.predicateOperatorType {

case .lessThan, .lessThanOrEqualTo:

    return .exactTime(.before, dateComponents)

case .greaterThan, .greaterThanOrEqualTo:

    return .exactTime(.after, dateComponents)

case .equalTo:

    return .exactTime(.at, dateComponents)

default:

    return nil

}

}

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 539 ังอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

/// - returns: The 'type' of HomeKit condition, with associated value, if applicable.

var homeKitConditionType: HomeKitConditionType {

    if let characteristic = characteristic() {

        return characteristic

    }

    else if let sunState = sunState() {

        return sunState

    }

    else if let exactTime = exactTime() {

        return exactTime

    }

    else {

        return .unknown

    }

}
}
}

```

UIColor+Custom

```

import UIKit

extension UIColor {

    /// - returns: A nice blue color which suggests editability.

    static let editableBlue = UIColor(red:0/255.0, green:122/255.0, blue:255/255.0,
alpha:1.0)
}

HMHome+Properties
import HomeKit

extension HMHome {

    /// All the services within all the accessories within the home.

    var allServices: [HMService] {

        return accessories.reduce([], { (accumulator, accessory) -> [HMService] in

            return accumulator + accessory.services.filter { return
!accumulator.contains($0) }

        })
    }
}

```

```

}

/// All the characteristics within all of the services within the home.

var allCharacteristics: [HMCharacteristic] {

    return allServices.reduce([], { (accumulator, service) -> [HMCharacteristic] in

        return accumulator + service.characteristics.filter { return

!accumulator.contains($0) }

    })

}

/**
- returns: A dictionary mapping localized service types to an array
of all services of that type.

*/

var serviceTable: [String: [HMService]] {

    var serviceDictionary = [String: [HMService]]()

    for service in self.allServices {

        if !service.isControlType {

```



```

        continue

    }

    let serviceType = service.localizedDescription

    var existingServices: [HMService] = serviceDictionary[serviceType] ??
[HMService]()

    existingServices.append(service)

    serviceDictionary[service.localizedDescription] = existingServices
}

for (serviceType, services) in serviceDictionary {

    serviceDictionary[serviceType] = services.sorted {

        return $0.accessory!.name.localizedCompare($1.accessory!.name) ==

.orderedAscending

    }

}

return serviceDictionary

}

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 543 ึงอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

/// - returns: All rooms in the home, including roomForEntireHome.

```
var allRooms: [HMRoom] {
```

```
    let allRooms = [self.roomForEntireHome()] + self.rooms
```

```
    return allRooms.sortByLocalizedName()
```

```
}
```

/// - returns: true if the current user is the admin of this home; false otherwise.

```
var isAdmin: Bool {
```

```
    return self.homeAccessControl(for: currentUser).isAdmin
```

```
}
```

/// - returns: All accessories which are 'control accessories'.

```
var sortedControlAccessories: [HMAccessory] {
```

```
    let filteredAccessories = self.accessories.filter { accessory -> Bool in
```

```
        for service in accessory.services {
```

```
            if service.isControlType {
```

```
                return true
```

```

    }

    }

    return false

}

return filteredAccessories.sortByLocalizedNames()

}

/**
- parameter identifier: The UUID to look up.
- returns: The accessory within the receiver that matches the given UUID,
or nil if there is no accessory with that UUID.

*/

func accessoryWithIdentifier(_ identifier: UUID) -> HMAccessory? {

    for accessory in self.accessories {

        if accessory.uniqueIdentifier == identifier {

            return accessory

        }

    }

```

```
}
```

```
return nil
```

```
}
```

```
/**
```

- parameter identifiers: An array of NSUUIDs that match accessories in the receiver.

- returns: An array of HMAccessory instances corresponding to the UUIDs passed in.

```
*/
```

```
func accessoriesWithIdentifiers(_ identifiers: [UUID]) -> [HMAccessory] {
```

```
    return self.accessories.filter { accessory in
```

```
        identifiers.contains(accessory.uniqueIdentifier)
```

```
    }
```

```
}
```

```
/**
```

Searches through the home's accessories to find the accessory

that is bridging the provided accessory.

- parameter accessory: The bridged accessory.

- returns: The accessory bridging the bridged accessory.

*/

```
func bridgeForAccessory(_ accessory: HMAccessory) -> HMAccessory? {  
    if !accessory.isBridged {  
        return nil  
    }  
  
    for bridge in self.accessories {  
        if let identifiers = bridge.uniqueIdentifiersForBridgedAccessories ,  
            identifiers.contains(accessory.uniqueIdentifier) {  
  
            return bridge  
  
        }  
  
    }  
  
    return nil
```

```

}

/**
 - parameter room: The room.

 - returns: The name of the room, appending "Default Room" if the room
            is the home's roomForEntireHome
 */
func nameForRoom(_ room: HMRoom) -> String {
    if room == self.roomForEntireHome() {
        let defaultRoom = NSLocalizedString("Default Room", comment: "Default
Room")
        return room.name + " \(defaultRoom)"
    }

    return room.name
}

/**

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 548 ังอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

- parameter zone: The zone.

- parameter rooms: A list of rooms to add to the final list.

- returns: A list of rooms that exist in the home and have not yet been added to this zone.

*/

```
func roomsNotAlreadyInZone(_ zone: HMZone, includingRooms rooms:
[HMRoom]? =
nil) -> [HMRoom] {
    var filteredRooms = self.rooms.filter { room in
        return !zone.rooms.contains(room)
    }

    if let rooms = rooms {
        filteredRooms += rooms
    }

    return filteredRooms
}
```

/**

- parameter home: The home.

- parameter serviceGroup: The service group.

- parameter services: A list of services to add to the final list.

- returns: A list of services that exist in the home and have not yet been added to this service group.

*/

```
func servicesNotAlreadyInServiceGroup(_ serviceGroup: HMServiceGroup,
includingServices services: [HMService]? = nil) -> [HMService] {
```

```
var filteredServices = self.allServices.filter { service in
```

```
/*
```

Exclude services that are already in the service group

and the accessory information service.

```
*/
```

```
return !serviceGroup.services.contains(service) && service.serviceType !=
```

```
HMServiceTypeAccessoryInformation
```

```

    }

    if let services = services {

        filteredServices += services

    }

    return filteredServices

}

}

UIAlertController+Convenience
import UIKit

extension UIAlertController {

/**

```

A simple UIAlertController that prompts for a name, then runs a completion block passing in

the name.

- parameter attributeType: The type of object that will be named.

- parameter completion: A block to call, passing in the provided text.
- parameter placeholder: An optional string used as text field's placeholder text.
- parameter shortType: An optional string used as to form the alert's action title.
- returns: A UIAlertController instance with a UITextField, cancel button, and add button.

```

*/
convenience init(attributeType: String, completionHandler: @escaping (_ name:
String) -> Void,
placeholder: String? = nil, shortType: String? = nil) {
    let title = NSLocalizedString("New", comment: "New") + " \(attributeType)"
    let message = NSLocalizedString("Enter a name.", comment: "Enter a name.")
    self.init(title: title, message: message, preferredStyle: .alert)
    self.addTextField { textField in
        textField.placeholder = placeholder ?? attributeType
        textField.autocapitalizationType = .words
    }
}

```

```

        let cancelAction = UIAlertAction(title: NSLocalizedString("Cancel", comment:
"Cancel"), style:

.cancel) { action in

        self.dismiss(animated: true, completion: nil)

    }

    let add = NSLocalizedString("Add", comment: "Add")

    let actionTitle = "\(add) \(shortType ?? attributeType)"

    let addNewObject = UIAlertAction(title: actionTitle, style: .default) { action in
        if let name = self.textFields!.first!.text {
            let trimmedName = name.trimmingCharacters(in:
CharacterSet.whitespacesAndNewlines)

            completionHandler(trimmedName)

        }

        self.dismiss(animated: true, completion: nil)

    }

    self.addAction(cancelAction)

    self.addAction(addNewObject)

}

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 553 ไปถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

/**

A simple UIAlertController made to show an error message that's passed in.

- parameter body: The body of the alert.

- returns: A UIAlertController with an 'Okay' button.

*/

```
convenience init(title: String, body: String) {  
    self.init(title: title, message: body, preferredStyle: .alert)  
    let okayAction = UIAlertAction(title: NSLocalizedString("Okay", comment:  
"Okay"), style:  
.default) { action in  
        self.dismiss(animated: true, completion: nil)  
    }  
    self.addAction(okayAction)  
}
```


CNMutablePostalAddress+Convenience

```
import MapKit
```

```
import Contacts
```

```
extension CNMutablePostalAddress {
```

```
    /// Constructs a CNMutablePostalAddress from a CLPlacemark
```

```
    convenience init(placemark: CLPlacemark) {
```

```
        self.init()
```

```
        self.street = (placemark.subThoroughfare ?? "") + " " + (placemark.thoroughfare  
?? "")
```

```
        self.city = placemark.locality ?? ""
```

```
        self.state = placemark.administrativeArea ?? ""
```

```
        self.postalCode = placemark.postalCode ?? ""
```

```
        self.country = placemark.country ?? ""
```

```
    }
```

```
}
```

HMActionSet+BuiltIn

```
import HomeKit
```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 555 ังอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

extension HMActionSet {

    struct Constants {

        static let builtInActionTypes = [HMActionSetTypeWakeUp,
HMActionSetTypeHomeDeparture, HMActionSetTypeHomeArrival,
HMActionSetTypeSleep]

    }

    /// - returns: true if the action set is built-in; false otherwise.
    var isBuiltIn: Bool {
        return Constants.builtInActionTypes.contains(self.actionSetType)
    }
}

UIViewController+Convenience

import HomeKit

import UIKit

extension UIViewController {

```

```
/**
```

Displays a UIAlertController on the main thread with the error's localizedDescription at the

body.

- parameter error: The error to display.

```
*/
```

```
func displayError(_ error: Error) {
```

```
    if error is CocoaError {
```

```
        let errorCode = HLError(_nsError: error as NSError).code
```

```
        if self.presentedViewController != nil || errorCode == .operationCancelled ||  
        errorCode ==
```

```
        .userDeclinedAddingUser {
```

```
            print(error.localizedDescription)
```

```
        }
```

```
    else {
```

```
        self.displayErrorMessage(error.localizedDescription)
```

```

    }

    }

    else {

        self.displayErrorMessage(String(describing: error))

    }

}

/**
Displays a collection of errors, separated by newlines.

- parameter errors: An array of NSError objects to display.

*/

func displayErrors(_ errors: [NSError]) {

    var messages = [String]()

    for error in errors {

        let errorCode = HLError(_nsError: error as NSError).code

        if self.presentedViewController != nil || errorCode == .operationCancelled ||
        errorCode ==

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 558 ังอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

.userDeclinedAddingUser {

    print(error.localizedDescription)

}

else {

    messages.append(error.localizedDescription)

}

}

if messages.count > 0 {

    // There were errors in the list, reduce the messages into a single one.

    let collectedMessage = messages.reduce("", { (accumulator, message) ->
String in
        return accumulator + "\n" + message

    })

    self.displayErrorMessage(collectedMessage)

}

}

```

```
/// Displays a UIAlertController with the passed-in text and an 'Okay' button.
```

```
func displayMessage(_ title: String, message: String) {
```

```
    DispatchQueue.main.async {
```

```
        let alert = UIAlertController(title: title, body: message)
```

```
        self.present(alert, animated: true, completion: nil)
```

```
    }
```

```
}
```

```
/**
```

Displays UIAlertController with a message and a localized "Error" title.

- parameter message: The message to display.

```
*/
```

```
private func displayErrorMessage(_ message: String) {
```

```
    let errorTitle = NSLocalizedString("Error", comment: "Error")
```

```
    displayMessage(errorTitle, message: message)
```

```
}
```



```
/**
```

Presents a simple UIAlertController with a textField, set up to

accept a name. Once the name is entered, the completion handler will

be called and the name will be passed in.

- parameter attributeType: The kind of object being added

- parameter completion: The block to run when the user taps the add button.

```
*/
```

```
func presentAddAlertWithAttributeType(_ type: String, placeholder: String? = nil,  
shortType:
```

```
String? = nil, completion: @escaping (String) -> Void) {
```

```
    let alertController = UIAlertController(attributeType: type, completionHandler:  
completion,
```

```
placeholder: placeholder, shortType: shortType)
```

```
    self.present(alertController, animated: true, completion: nil)
```

```
}
```

```
}
```

Array+Sorting

import HomeKit

/// A protocol for objects which have a property called name.

protocol Nameable {

var name: String { get }

}

/*

All of these HomeKit objects have names and can conform
to this protocol without modification.

*/

extension HMHome: Nameable {}

extension HMAccessory: Nameable {}

extension HMRoom: Nameable {}

extension HMZone: Nameable {}

extension HMActionSet: Nameable {}

```
extension HMService: Nameable {}
```

```
extension HMServiceGroup: Nameable {}
```

```
extension HMTrigger: Nameable {}
```

```
extension Collection where Iterator.Element: Nameable {
```

```
  /**
```

Generates a new array from the original collection,
sorted by localized name.

- returns: New array sorted by localized name.

```
  */
```

```
  func sortByLocalizedNames() -> [Iterator.Element] {
```

```
    return sorted { $0.name.localizedCompare($1.name) ==  
      .orderedAscending }
```

```
  }
```

```
}
```

```
extension Collection where Iterator.Element: HMAActionSet {
```

/**

Generates a new array from the original collection,

sorted by built-in first, then user-defined sorted

by localized name.

- returns: New array sorted by localized name.

*/

```
func sortByTypeAndLocalizedName() -> [HMAActionSet] {
```

```
    return sorted { (actionSet1, actionSet2) -> Bool in
```

```
        if actionSet1.isBuiltIn != actionSet2.isBuiltIn {
```

```
            // If comparing a built-in and a user-defined, the built-in is ranked first.
```

```
            return actionSet1.isBuiltIn
```

```
        }
```

```
        else if actionSet1.isBuiltIn && actionSet2.isBuiltIn {
```

```
            // If comparing two built-ins, we follow a standard ranking
```

```
            let firstIndex =
```

```
HMAActionSet.Constants.builtInActionSetTypes.firstIndex(of:
```

```
actionSet1.actionSetType) ?? NSNotFound
```

```

let secondIndex =
HMActionSet.Constants.builtInActionTypes.firstIndex(of:
actionSet2.actionSetType) ?? NSNotFound

return firstIndex < secondIndex

}

else {

// If comparing two user-defines, sort by localized name.
return actionSet1.name.localizedCompare(actionSet2.name) ==
.orderedAscending
}
}
}

}

}

HMCharacteristic+Properties

import HomeKit

extension HMCharacteristic {

```

```

private struct Constants {

    static let valueFormatter = NumberFormatter()

    static let numericFormats = [

        HMCharacteristicMetadataFormatInt,

        HMCharacteristicMetadataFormatFloat,

        HMCharacteristicMetadataFormatUInt8,

        HMCharacteristicMetadataFormatUInt16,

        HMCharacteristicMetadataFormatUInt32,

        HMCharacteristicMetadataFormatUInt64

    ]

}

/**

```

Returns the localized description for a provided value, taking the characteristic's metadata

and possible

values into account.

- parameter value: The value to look up.

- returns: A string representing the value in a localized way, e.g. "24%" or "354°"

*/

```
func localizedDescriptionForValue(_ value: AnyObject) -> String {  
  
    if self.isWriteOnly {  
        return NSLocalizedString("Write-Only Characteristic", comment: "Write-  
Only  
Characteristic")  
    }  
  
    else if self.isBoolean {  
        if let boolValue = value.boolValue {  
            return boolValue ? NSLocalizedString("On", comment: "On") :  
NSLocalizedString("Off",  
comment: "Off")  
        }  
    }  
  
    if let number = value as? Int {
```

```

        if let predeterminedValueString =
self.predeterminedValueDescriptionForNumber(number)
{
    return predeterminedValueString
}

if let stepValue = self.metadata?.stepValue {
    Constants.valueFormatter.minimumFractionDigits = Int(log10(1.0 /
stepValue.doubleValue))
    if let string = Constants.valueFormatter.string(from: NSNumber(value:
number)) {
        return string + self.localizedUnitDecoration
    }
}

return "\(value)"
}

```

/**

- parameter number: The value of this characteristic.

- returns: An optional, localized string for the value.

*/

```
func predeterminedValueDescriptionForNumber(_ number: Int) -> String? {  
  
    switch self.characteristicType {  
  
        case HMCharacteristicTypePowerState, HMCharacteristicTypeInputEvent,  
HMCharacteristicTypeOutputState:  
  
            if number != 0 {  
  
                return NSLocalizedString("On", comment: "On")  
  
            }  
  
            else {  
  
                return NSLocalizedString("Off", comment: "Off")  
  
            }  
  
        case HMCharacteristicTypeOutletInUse,  
HMCharacteristicTypeMotionDetected,
```

HMCharacteristicTypeAdminOnlyAccess, HMCharacteristicTypeAudioFeedback,

HMCharacteristicTypeObstructionDetected:

```
if number != 0 {
```

```
    return NSLocalizedString("Yes", comment: "Yes")
```

```
}
```

```
else {
```

```
    return NSLocalizedString("No", comment: "No")
```

```
}
```

```
case
```

HMCharacteristicTypeTargetDoorState,

HMCharacteristicTypeCurrentDoorState:

```
if let doorState = HMCharacteristicValueDoorState(rawValue: number) {
```

```
    switch doorState {
```

```
        case .open:
```

```
            return NSLocalizedString("Open", comment: "Open")
```

```
        case .opening:
```

```
            return NSLocalizedString("Opening", comment: "Opening")
```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 570 ไปถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```
case .closed:
```

```
    return NSLocalizedString("Closed", comment: "Closed")
```

```
case .closing:
```

```
    return NSLocalizedString("Closing", comment: "Closing")
```

```
case .stopped:
```

```
    return NSLocalizedString("Stopped", comment: "Stopped")
```

```
@unknown default:
```

```
    return ""
```

```
}
```

```
}
```

```
case HMCharacteristicTypeTargetHeatingCooling:
```

```
    if let mode = HMCharacteristicValueHeatingCooling(rawValue: number)
```

```
{
```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 571 ไปถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

switch mode {

    case .off:

        return NSLocalizedString("Off", comment: "Off")

    case .heat:

        return NSLocalizedString("Heat", comment: "Heat")

    case .cool:

        return NSLocalizedString("Cool", comment: "Cool")

    case .auto:

        return NSLocalizedString("Auto", comment: "Auto")

    @unknown default:

        return ""

}

}

```

case HMCharacteristicTypeCurrentHeatingCooling;


```

if let mode = HMCharacteristicValueHeatingCooling(rawValue: number)
{

    switch mode {

        case .off:

            return NSLocalizedString("Off", comment: "Off")

        case .heat:

            return NSLocalizedString("Heating", comment: "Heating")

        case .cool:

            return NSLocalizedString("Cooling", comment: "Cooling")

        case .auto:

            return NSLocalizedString("Auto", comment: "Auto")

        @unknown default:

            return ""

    }

}

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 573 ไปถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

case HMCharacteristicTypeTargetLockMechanismState,
HMCharacteristicTypeCurrentLockMechanismState:

    if let lockState = HMCharacteristicValueLockMechanismState(rawValue:
number) {

        switch lockState {

            case .unsecured:

                return NSLocalizedString("Unsecured", comment: "Unsecured")

            case .secured:

                return NSLocalizedString("Secured", comment: "Secured")

            case .unknown:

                return NSLocalizedString("Unknown", comment: "Unknown")

            case .jammed:

                return NSLocalizedString("Jammed", comment: "Jammed")

```

```

@unknown default:

    return ""

}

}

case HMCharacteristicTypeTemperatureUnits:

    if let unit = HMCharacteristicValueTemperatureUnit(rawValue: number)
    {
        switch unit {
        case .celsius:
            return NSLocalizedString("Celsius", comment: "Celsius")

        case .fahrenheit:
            return NSLocalizedString("Fahrenheit", comment: "Fahrenheit")

        @unknown default:

            return ""

        }
    }
}

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 575 ไปถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

    }

    case HMCharacteristicTypeLockMechanismLastKnownAction:

        if let lastKnownAction =

HMCharacteristicValueLockMechanismLastKnownAction(rawValue: number) {

            switch lastKnownAction {

                case .securedUsingPhysicalMovementInterior:

                    return NSLocalizedString("Interior Secured", comment: "Interior
Secured")

                case .unsecuredUsingPhysicalMovementInterior:

                    return NSLocalizedString("Exterior Unsecured", comment:
"Exterior Unsecured")

                case .securedUsingPhysicalMovementExterior:

                    return NSLocalizedString("Exterior Secured", comment: "Exterior
Secured")

                case .unsecuredUsingPhysicalMovementExterior:

```

```
return NSLocalizedString("Exterior Unsecured", comment:  
"Exterior Unsecured")
```

```
case .securedWithKeypad:
```

```
return NSLocalizedString("Keypad Secured", comment: "Keypad  
Secured")
```

```
case .unsecuredWithKeypad:
```

```
return NSLocalizedString("Keypad Unsecured", comment:  
"Keypad Unsecured")
```

```
case .securedRemotely:
```

```
return NSLocalizedString("Secured Remotely", comment:  
"Secured Remotely")
```

```
case .unsecuredRemotely:
```

```
return NSLocalizedString("Unsecured Remotely", comment:  
"Unsecured  
Remotely")
```

```

        case .securedWithAutomaticSecureTimeout:

            return NSLocalizedString("Secured Automatically", comment:
"Secured
Automatically")

        case .securedUsingPhysicalMovement:

            return NSLocalizedString("Secured Using Physical Movement",
comment: "Secured
Using Physical Movement")

        case .unsecuredUsingPhysicalMovement:

            return NSLocalizedString("Unsecured Using Physical Movement",
comment:
"Unsecured Using Physical Movement")

        @unknown default:

            return ""

    }

}

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 578 จึงอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้


```

case HMCharacteristicTypeRotationDirection:

    if let rotationDirection =
        HMCharacteristicValueRotationDirection(rawValue: number) {

        switch rotationDirection {

            case .clockwise:

                return NSLocalizedString("Clockwise", comment: "Clockwise")

            case .counterClockwise:

                return NSLocalizedString("Counter Clockwise", comment:
                    "Counter Clockwise")

            @unknown default:

                return ""

        }

    }

case HMCharacteristicTypeAirParticulateSize:

```

```

if let size = HMCharacteristicValueAirParticulateSize(rawValue: number)
{

    switch size {

        case .size10:

            return NSLocalizedString("Size 10", comment: "Size 10")

        case .size2_5:

            return NSLocalizedString("Size 2.5", comment: "Size 2.5")

        @unknown default:

            return ""

    }

}

case HMCharacteristicTypePositionState:

    if let state = HMCharacteristicValuePositionState(rawValue: number) {

        switch state {

            case .opening:

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 580 ไปถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```
return NSLocalizedString("Opening", comment: "Opening")
```

```
case .closing:
```

```
return NSLocalizedString("Closing", comment: "Closing")
```

```
case .stopped:
```

```
return NSLocalizedString("Stopped", comment: "Stopped")
```

```
@unknown default:
```

```
return ""
```

```
}
```

```
}
```

```
case HMCharacteristicTypeCurrentSecuritySystemState:
```

```
if let state =
```

```
HMCharacteristicValueCurrentSecuritySystemState(rawValue: number) {
```

```
switch state {
```

```
case .awayArm:
```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 581 ไปถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```
return NSLocalizedString("Away", comment: "Away")
```

```
case .stayArm:
```

```
return NSLocalizedString("Home", comment: "Home")
```

```
case .nightArm:
```

```
return NSLocalizedString("Night", comment: "Night")
```

```
case .disarmed:
```

```
return NSLocalizedString("Disarm", comment: "Disarm")
```

```
case .triggered:
```

```
return NSLocalizedString("Triggered", comment: "Triggered")
```

```
@unknown default:
```

```
return ""
```

```
}
```

```
}
```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 582 ไปถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

case HMCharacteristicTypeTargetSecuritySystemState:

    if let state =
        HMCharacteristicValueTargetSecuritySystemState(rawValue: number) {

        switch state {

            case .awayArm:

                return NSLocalizedString("Away", comment: "Away")

            case .stayArm:

                return NSLocalizedString("Home", comment: "Home")

            case .nightArm:

                return NSLocalizedString("Night", comment: "Night")

            case .disarm:

                return NSLocalizedString("Disarm", comment: "Disarm")

            @unknown default:

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 583 ึงอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

        return ""

    }

}

default:

    break

}

return nil
}

var supportsEventNotification: Bool {

    return

self.properties.contains(NSNotification.Name.HMCharacteristicPropertySupportsEvent.rawValue)

}

/// - returns: A string representing the value in a localized way, e.g. "24%" or
"354°"

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 584 ังอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้


```

var localizedValueDescription: String {

    if let value = value {

        return self.localizedDescriptionForValue(value as AnyObject)

    }

    return ""

}

/// - returns: The decoration for the characteristic's units, localized, e.g. "%" or "°"
var localizedUnitDecoration: String {

    if let units = self.metadata?.units {

        switch units {

        case HMCharacteristicMetadataUnitsCelsius:

            return NSLocalizedString("°C", comment: "Degrees Celsius")

        case HMCharacteristicMetadataUnitsArcDegree:

            return NSLocalizedString("°", comment: "Arc Degrees")

        case HMCharacteristicMetadataUnitsFahrenheit:

```

```

        return NSLocalizedString("°F", comment: "Degrees Fahrenheit")

    case HMCharacteristicMetadataUnitsPercentage:

        return NSLocalizedString("%", comment: "Percentage")

    default: break
    }
}

return ""
}

/// - returns: The type of the characteristic, e.g. "Current Lock Mechanism State"

var localizedCharacteristicType: String {

    var type = self.localizedDescription

    var localizedDescription: String? = nil

    if isReadOnly {

```

```

        localizedDescription = NSLocalizedString("Read Only", comment: "Read
Only")

    }

    else if isWriteOnly {

        localizedDescription = NSLocalizedString("Write Only", comment: "Write
Only")

    }

    if let localizedDescription = localizedDescription {

        type = type + " \(localizedDescription)"

    }

    return type

}

/// - returns: true if this characteristic has numeric values that are all integers; false
otherwise.

//  var isInteger: Bool {

//      return self.isNumeric && !self.isFloatingPoint

```

```

// }

/**

- returns: true if this characteristic has numeric values;

false otherwise.

*/

// var isNumeric: Bool {
//   guard let metadata = metadata else { return false }
//   guard let format = metadata.format else { return false }
//   return Constants.numericFormats.contains(format)
// }

/// - returns: true if this characteristic is boolean; false otherwise.

var isBoolean: Bool {

  guard let metadata = metadata else { return false }

  return metadata.format == HMCharacteristicMetadataFormatBool

}

```

```

/**

- returns: true if this characteristic is text-writable;

false otherwise.

*/

var isTextWritable: Bool {

guard let metadata = metadata else { return false }

return metadata.format == HMCharacteristicMetadataFormatString &&
properties.contains(HMCharacteristicPropertyWritable)

}

/**

- returns: true if this characteristic has numeric values

that are all floating point; false otherwise.

*/

var isFloatingPoint: Bool {

guard let metadata = metadata else { return false }

return metadata.format == HMCharacteristicMetadataFormatFloat

}

```

/// - returns: true if characteristic is read only; false otherwise.

var isReadOnly: Bool {

return !properties.contains(HMCharacteristicPropertyWritable) &&

properties.contains(HMCharacteristicPropertyReadable)

}

/**

- returns: true if this characteristic is write only;

false otherwise.

*/

var isWriteOnly: Bool {

return !properties.contains(HMCharacteristicPropertyReadable) &&

properties.contains(HMCharacteristicPropertyWritable)

}

/**

- returns: true if this characteristic is the 'Identify'

characteristic; false otherwise.

*/

var isIdentify: Bool {

return self.characteristicType == HMCharacteristicTypeIdentify

}

/**

- returns: The number of possible values that this characteristic can contain.

The standard formula for the number of values between two numbers is

((greater - lesser) + 1), and this takes step value into account.

*/

var numberOfChoices: Int {

guard let metadata = metadata, let minimumValue = metadata.minimumValue
as? Int else {

return 0 }

guard let maximumValue = metadata.maximumValue as? Int else { return 0 }

var range = maximumValue - minimumValue

if let stepValue = metadata.stepValue as? Double {

```

        range = Int(Double(range) / stepValue)

    }

    return range + 1

}

/// - returns: All of the possible values that this characteristic can contain.

var allPossibleValues: [AnyObject]? {

//    guard self.isInteger else { return nil }

    guard let metadata = metadata, let stepValue = metadata.stepValue as? Double
    else { return
nil }

    let choices = Array(0..

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 592 ไปถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

- returns: true if this characteristic has value descriptions separate from just displaying

raw values, e.g. Secured or Jammed; false otherwise.

*/

```
var hasPredeterminedValueDescriptions: Bool {
```

```
    guard let number = self.value as? Int else { return false }
```

```
    return self.predeterminedValueDescriptionForNumber(number) != nil
```

```
}
```

```
}
```

HMService+Properties

```
import HomeKit
```

```
extension HMService {
```

```
    struct Constants {
```

```
        static let serviceMap = [
```

```
            HMServiceTypeLightbulb: NSLocalizedString("Lightbulb", comment: "Lightbulb"),
```

```
            HMServiceTypeFan: NSLocalizedString("Fan", comment: "Fan")
```

```

    ]

}

/**
    - parameter serviceType: The service type.

    - returns: A localized description of that service type or
               the original type string if one cannot be found.
 */
class func localizedDescriptionForServiceType(_ type: String) -> String {
    return Constants.serviceMap[type] ?? type
}

/// - returns: true if this service supports the associatedServiceType property; false
otherwise.

var supportsAssociatedServiceType: Bool {

    return self.serviceType == HMServiceTypeOutlet || self.serviceType ==
    HMServiceTypeSwitch

```

```

    }

    /// - returns: true if this service is a 'control type'; false otherwise.

    var isControlType: Bool {

        let noncontrolTypes = [HMServiceTypeAccessoryInformation,
HMServiceTypeLockManagement]

        return !noncontrolTypes.contains(self.serviceType)
    }

    /**
    - returns: The valid associated service types for this service,
    e.g. HMServiceTypeFan or HMServiceTypeLightbulb

    */

    class var validAssociatedServiceTypes: [String] {

        return [HMServiceTypeFan, HMServiceTypeLightbulb]

    }

}

```

RoomViewController

```
import UIKit

import HomeKit

/// A view controller that lists the accessories within a room.

class RoomViewController: HMCatalogViewController, HMAccessoryDelegate {

    // MARK: Types

    struct Identifiers {

        static let accessoryCell = "AccessoryCell"

        static let unreachableAccessoryCell = "UnreachableAccessoryCell"

        static let modifyAccessorySegue = "Modify Accessory"

    }

    // MARK: Properties

    var room: HMRoom! {

        didSet {

            navigationItem.title = room.name
```



```

    }

}

var accessories = [HMAccessory]()

// MARK: View Methods

override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)
    reloadData()
}

// MARK: Table View Methods

/// - returns: The number of accessories within this room.

override func tableView(_ tableView: UITableView, numberOfRowsInSectionSection
section: Int) -> Int {

    let rows = accessories.count

```

```

if rows == 0 {

    let message = NSLocalizedString("No Accessories", comment: "No
Accessories")

    setBackgroundMessage(message)

}

else {

    setBackgroundMessage(nil)

}

return rows

}

/// - returns: true if the current room is not the home's roomForEntireHome; false
otherwise.

override func tableView(_ tableView: UITableView, canEditRowAt indexPath:
IndexPath) -> Bool

{

    return room != home.roomForEntireHome()

}

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 598 ไปถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

/// - returns: Localized "Unassign".

override func tableView(_ tableView: UITableView,
titleForDeleteConfirmationButtonForRowAt
indexPath: IndexPath) -> String? {

    return NSLocalizedString("Unassign", comment: "Unassign")
}

/// Assigns the 'deleted' room to the home's roomForEntireHome.
override func tableView(_ tableView: UITableView, commit editingStyle:
UITableViewCellEditingStyle, forRowAt indexPath: IndexPath) {
    if editingStyle == .delete {
        unassignAccessory(accessories[indexPath.row])
    }
}

/// - returns: A cell representing an accessory.

```

```

    override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) ->

UITableViewCell {

    let accessory = accessories[indexPath.row]

    var reuseIdentifier = Identifiers.accessoryCell

    if !accessory.isReachable {
        reuseIdentifier = Identifiers.unreachableAccessoryCell
    }

    let cell = tableView.dequeueReusableCell(withIdentifier: reuseIdentifier, for:
indexPath)

    cell.textLabel?.text = accessory.name

    return cell

}

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 600 ไปถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

/// - returns: A localized description, "Accessories" if there are accessories to list.

override func tableView(_ tableView: UITableView, titleForHeaderInSection
section: Int) ->

String? {

    if accessories.isEmpty {

        return nil

    }

    return NSLocalizedString("Accessories", comment: "Accessories")

}

// MARK: Helper Methods

/// Updates the internal array of accessories and reloads the table view.

private func reloadData() {

    accessories = room.accessories.sortByLocalizedString()

    tableView.reloadData()

```

```

}

/// Sorts the internal list of accessories by localized name.

private func sortAccessories() {

    accessories = accessories.sortByLocalizedName()

}

/**
Registers as the delegate for the current home and
all accessories in our room.
*/

override func registerAsDelegate() {

    super.registerAsDelegate()

    for accessory in room.accessories {

        accessory.delegate = self

    }

}

```



```
/// Sets the accessory and home of the modifyAccessoryViewController that will  
be presented.
```

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
```

```
    super.prepare(for: segue, sender: sender)
```

```
    let indexPath = tableView.indexPath(for: sender as! UITableViewCell)!
```

```
    if segue.identifier == Identifiers.modifyAccessorySegue {
```

```
        let modifyViewController = segue.intendedDestinationViewController as!
```

```
        ModifyAccessoryViewController
```

```
        modifyViewController.accessory = room.accessories[indexPath.row]
```

```
    }
```

```
}
```

```
/**
```

Adds an accessory into the internal list of accessories

and inserts the row into the table view.

- parameter accessory: The HMAccessory to add.

```
*/
```

```

private func didAssignAccessory(_ accessory: HMAccessory) {

    accessories.append(accessory)

    sortAccessories()

    if let newAccessoryIndex = accessories.firstIndex(of: accessory) {

        let newAccessoryIndexPath = IndexPath(row: newAccessoryIndex, section:
0)

        tableView.insertRows(at: [newAccessoryIndexPath], with: .automatic)
    }
}

/**
Removes an accessory from the internal list of accessory (if it
exists) and deletes the row from the table view.

- parameter accessory: The HMAccessory to remove.

*/

private func didUnassignAccessory(_ accessory: HMAccessory) {

    if let accessoryIndex = accessories.firstIndex(of: accessory) {

```

```

accessories.remove(at: accessoryIndex)

let accessoryIndexPath = IndexPath(row: accessoryIndex, section: 0)

tableView.deleteRows(at: [accessoryIndexPath], with: .automatic)

}

}

/**
Assigns an accessory to the current room.
- parameter accessory: The HMAccessory to assign to the room.
*/
private func assignAccessory(_ accessory: HMAccessory) {

    didAssignAccessory(accessory)

    home.assignAccessory(accessory, to: room) { error in

        if let error = error {

            self.displayError(error)

            self.didUnassignAccessory(accessory)

}

```

```

    }

}

/**

Assigns the current room back into roomForEntireHome.

- parameter accessory: The HMAccessory to reassign.

*/
private func unassignAccessory(_ accessory: HMAccessory) {
    didUnassignAccessory(accessory)

    home.assignAccessory(accessory, to: home.roomForEntireHome()) { error in
        if let error = error {

            self.displayError(error)

            self.didAssignAccessory(accessory)

        }

    }

}

}

}

```

```
/**
```

Finds an accessory in the internal array of accessories

and updates its row in the table view.

- parameter accessory: The HMAccessory to reload.

```
*/
```

```
func didModifyAccessory(_ accessory: HMAccessory){
```

```
    if let index = accessories.firstIndex(of: accessory) {
```

```
        let indexPaths = [
```

```
            IndexPath(row: index, section: 0)
```

```
        ]
```

```
        tableView.reloadRows(at: indexPaths, with: .automatic)
```

```
    }
```

```
}
```

```
// MARK: HMHomeDelegate Methods
```

/// If the accessory was added to this room, insert it.

```
func home(_ home: HMHome, didAddAccessory accessory: HMAccessory) {
```

```
    if accessory.room == room {
```

```
        accessory.delegate = self
```

```
        didAssignAccessory(accessory)
```

```
    }
```

```
}
```

/// Remove the accessory from our room, if required.

```
func home(_ home: HMHome, didRemoveAccessory accessory: HMAccessory) {
```

```
    didUnassignAccessory(accessory)
```

```
}
```

/**

Handles the update.

We act based on one of three options:

1. A new accessory is being added to this room.
2. An accessory is being assigned from this room to another room.
3. We can ignore this message.

*/

```
func home(_ home: HMHome, didUpdateRoom room: HMRoom, forAccessory  
accessory:
```

```
HMAccessory) {
```

```
    if room == self.room {
```

```
        didAssignAccessory(accessory)
```

```
    }
```

```
    else if accessories.contains(accessory) {
```

```
        didUnassignAccessory(accessory)
```

```
    }
```

```
}
```

```
/// If our room was removed, pop back.
```

```
func home(_ home: HMHome, didRemoveRoom room: HMRoom) {
```

```
    if room == self.room {
```

```

navigationController!.popViewController(animated: true)

}

}

/// If our room was renamed, reload our title.

func home(_ home: HMHome, didUpdateNameFor room: HMRoom) {

    if room == self.room {

        navigationItem.title = room.name

    }

}

// MARK: HMAccessoryDelegate Methods

// Accessory updates will reload the cell for the accessory.

func accessoryDidUpdateReachability(_ accessory: HMAccessory) {

    didModifyAccessory(accessory)

```

```

    }

    func accessoryDidUpdateName(_ accessory: HMAccessory) {

        didModifyAccessory(accessory)

    }

}

HomeKitObjectCollection

import HomeKit

/// Represents the all different types of HomeKit objects.

enum HomeKitObjectSection: Int {

    case actionSet, trigger

    static let count = 2

}

/**

```

Manages internal lists of HomeKit objects to allow for

save insertion into a table view.

*/

```
class HomeKitObjectCollection {
```

```
    // MARK: Properties
```

```
    // var accessories = [HMAccessory]()
```

```
    //var rooms = [HMRoom]()
```

```
    //var zones = [HMZone]()
```

```
    var actionSets = [HMActionSet]()
```

```
    var triggers = [HMTrigger]()
```

```
    //var serviceGroups = [HMServiceGroup]()
```

```
    /**
```

Adds an object to the collection by finding its corresponding

array and appending the object to it.

- parameter object: The HomeKit object to append.

```
    */
```

```

func append(_ object: AnyObject) {

    switch object {

        case let actionSet as HMActionSet:

            actionSets.append(actionSet)

            actionSets = actionSets.sortByTypeAndLocalizedName()

//         case let accessory as HMAccessory:
//
//             accessories.append(accessory)
//
//             accessories = accessories.sortByLocalizedName()

//         case let room as HMRoom:
//
//             rooms.append(room)
//
//             rooms = rooms.sortByLocalizedName()

//         case let zone as HMZone:
//
//             zones.append(zone)
//
//             zones = zones.sortByLocalizedName()

```

```
case let trigger as HMTrigger:
```

```
    triggers.append(trigger)
```

```
    triggers = triggers.sortByLocalizedName()
```

```
// case let serviceGroup as HMServiceGroup:
```

```
//     serviceGroups.append(serviceGroup)
```

```
//     serviceGroups = serviceGroups.sortByLocalizedName()
```

```
    default:
```

```
        break
```

```
    }
```

```
}
```

```
/**
```

Creates an NSIndexPath representing the location of the

HomeKit object in the table view.

- parameter object: The HomeKit object to find.

- returns: The NSIndexPath representing the location of
the HomeKit object in the table view.

*/

```
func indexPathOfObject(_ object: AnyObject) -> IndexPath? {  
    switch object {  
    case let actionSet as HMActionSet:  
        if let index = actionSets.firstIndex(of: actionSet) {  
            return IndexPath(row: index,  
HomeKitObjectSection.actionSet.rawValue)  
        }  
    // case let accessory as HMAccessory:  
    //     if let index = accessories.firstIndex(of: accessory) {  
    //         return IndexPath(row: index, section:  
HomeKitObjectSection.accessory.rawValue)  
    //     }  
}
```

```

//      case let room as HMRoom:

//          if let index = rooms.firstIndex(of: room) {

//              return IndexPath(row: index, section:
HomeKitObjectSection.room.rawValue)

//          }

//      case let zone as HMZone:

//          if let index = zones.firstIndex(of: zone) {

//              return IndexPath(row: index, section:
HomeKitObjectSection.zone.rawValue)

//          }

//      case let trigger as HMTrigger:

//          if let index = triggers.firstIndex(of: trigger) {

//              return IndexPath(row: index, section:
HomeKitObjectSection.trigger.rawValue)

//          }

//      case let serviceGroup as HMServiceGroup:

```

```
//      if let index = serviceGroups.firstIndex(of: serviceGroup) {

//                               return IndexPath(row: index, section:
HomeKitObjectSection.serviceGroup.rawValue)

//      }
```

```
        default: break
```

```
    }
```

```
    return nil
```

```
}
```

```
/**
```

Removes a HomeKit object from the collection.

- parameter object: The HomeKit object to remove.

```
*/
```

```
func remove(_ object: AnyObject) {
```

```
    switch object {
```

```

case let actionSet as HMActionSet:

    if let index = actionSets.firstIndex(of: actionSet) {

        actionSets.remove(at: index)

    }

//
// case let accessory as HMAccessory:
//
//     if let index = accessories.firstIndex(of: accessory) {
//
//         accessories.remove(at: index)
//
//     }
//
// case let room as HMRoom:
//
//     if let index = rooms.firstIndex(of: room) {
//
//         rooms.remove(at: index)
//
//     }
//
//
// case let zone as HMZone:
//
//     if let index = zones.firstIndex(of: zone) {
//
//         zones.remove(at: index)

```

```

//      }

case let trigger as HMTrigger:

    if let index = triggers.firstIndex(of: trigger) {

        triggers.remove(at: index)

    }

//      case let serviceGroup as HMServiceGroup:
//      if let index = serviceGroups.firstIndex(of: serviceGroup) {
//      serviceGroups.remove(at: index)
//      }

default:

    break

    }

}

```

/**

Provides the array of NSObjects corresponding to the provided section.

- parameter section: A HomeKitObjectSection.

- returns: An array of NSObjects corresponding to the provided section.

*/

```
func objectsForSection(_ section: HomeKitObjectSection) -> [NSObject] {
```

```
    switch section {
```

```
        case .actionSet:
```

```
            return actionSets
```

```
        // case .accessory:
```

```
        //     return accessories
```

```
        // case .room:
```

```
        //     return rooms
```

```
        // case .zone:
```



```

//      return zones

    case .trigger:

        return triggers

//
//      case .serviceGroup:
//
//      return serviceGroups

    default:
        return []
    }
}

/**

```

Provides an HomeKitObjectSection for a given object.

- parameter object: A HomeKit object.

- returns: The corresponding HomeKitObjectSection

*/

```
class func sectionForObject(_ object: AnyObject?) -> HomeKitObjectSection? {
```

```
    switch object {
```

```
        case is HMActionSet:
```

```
            return .actionSet
```

```
//        case is HMAccessory:
```

```
//            return .accessory
```

```
//        case is HMZone:
```

```
//            return .zone
```

```
//        case is HMRoom:
```

```
//            return .room
```

```
        case is HMTrigger:
```

```
            return .trigger
```

```
// case is HMServiceGroup:
```

```
// return .serviceGroup
```

```
default:
```

```
return nil
```

```
}
```

```
}
```

```
/**
```

Reloads all internal structures based on the provided home.

- parameter home: The HMHome with which to reset the collection.

```
*/
```

```
func resetWithHome(_ home: HMHome) {
```

```
// accessories = home.accessories.sortByLocalizedName()
```

```
// rooms = home.allRooms
```

```
// zones = home.zones.sortByLocalizedName()
```

```

    actionSets = home.actionSets.sortByTypeAndLocalizedString()

    triggers = home.triggers.sortByLocalizedString()

//    serviceGroups = home.serviceGroups.sortByLocalizedString()

}

}

ActionSetViewController

import UIKit
import HomeKit

/// Represents table view sections of the ActionSetViewController.
enum ActionSetTableViewSection: Int {

    case name, actions, accessories

}

```

/**

A view controller that facilitates creation of Action Sets.

It contains a cell for a name, and lists accessories within a home.

If there are actions within the action set, it also displays a list of ActionCells displaying those

actions.

It owns an ActionSetCreator and routes events to the creator as appropriate.

*/

```
class ActionSetViewController: HMCatalogViewController {
```

```
    // MARK: Types
```

```
    struct Identifiers {
```

```
        static let accessoryCell = "AccessoryCell"
```

```
        static let unreachableAccessoryCell = "UnreachableAccessoryCell"
```

```
        static let actionCell = "ActionCell"
```

```
        static let showServiceSegue = "Show Services"
```

```
    }
```

```
    // MARK: Properties
```

```
    @IBOutlet weak var nameField: UITextField!
```

```
@IBOutlet weak var saveButton: UIBarButtonItem!
```

```
var actionSet: HMActionSet?
```

```
var actionSetCreator: ActionSetCreator!
```

```
var displayedAccessories = [HMAccessory]()
```

```
// MARK: View Methods
```

```
/**
```

Creates the action set creator, registers the appropriate reuse identifiers in the table,

and sets the nameField if appropriate.

```
*/
```

```
override func viewDidLoad() {
```

```
    super.viewDidLoad()
```

```
    actionSetCreator = ActionSetCreator(actionSet: actionSet, home: home)
```

```
    displayedAccessories = home.sortedControlAccessories
```



```
tableView.register(UITableViewCell.self, forCellReuseIdentifier:
Identifiers.accessoryCell)
```

```
tableView.register(UITableViewCell.self, forCellReuseIdentifier:
Identifiers.unreachableAccessoryCell)
```

```
tableView.register(ActionCell.self, forCellReuseIdentifier:
Identifiers.actionCell)
```

```
tableView.rowHeight = UITableView.automaticDimension
```

```
tableView.estimatedRowHeight = 44.0
```

```
if let actionSet = actionSet {
    nameField.text = actionSet.name
    nameFieldDidChange(nameField)
}
```

```
if !home.isAdmin {
```

```
    nameField.isEnabled = false
```

```

    }

}

/// Reloads the data and view.

override func viewWillAppear(_ animated: Bool) {

    super.viewWillAppear(animated)

    tableView.reloadData()

    enableSaveButtonIfNecessary()
}

/// Dismisses the view controller if our data is invalid.

override func viewDidAppear(_ animated: Bool) {

    super.viewDidAppear(animated)

    if shouldPopViewController() {

        self.dismiss(animated: true, completion: nil)

    }

}

```

```

/// Dismisses the keyboard when we dismiss.

override func viewWillDisappear(_ animated: Bool) {

    super.viewWillDisappear(animated)

    resignFirstResponder()

}

/// Passes our accessory into the ServicesViewController.

override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    super.prepare(for: segue, sender: sender)
    if segue.identifier == Identifiers.showServiceSegue {
        let servicesViewController = segue.intendedDestinationViewController as!
ServicesViewController

        servicesViewController.onlyShowsControlServices = true

        servicesViewController.cellDelegate = actionSetCreator

        let index = tableView.indexPath(for: sender as! UITableViewCell)?.row

        servicesViewController.accessory = displayedAccessories[index]

```

```

servicesViewController.cellDelegate = actionSetCreator

}

}

// MARK: IBAction Methods

/// Dismisses the view controller.
@IBAction func dismiss() {
    self.dismiss(animated: true, completion: nil)
}

/// Saves the action set, adds it to the home, and dismisses the view.
@IBAction func saveAndDismiss() {

    saveButton.isEnabled = false

    actionSetCreator.saveActionSetWithName(trimmedName) { error in

        self.saveButton.isEnabled = true

```

```

        if let error = error {

            self.displayError(error)

        }

        else {

            self.dismiss()

        }

    }

}

/// Prompts an update to the save button enabled state.

@IBAction func nameFieldDidChange(_ field: UITextField) {

    enableSaveButtonIfNecessary()

}

// MARK: Table View Methods

```

```

/// We do not allow the creation of action sets in a shared home.

```

```
override func numberOfSections(in tableView: UITableView) -> Int {
```

```
    return home.isAdmin ? 3 : 2
```

```
}
```

```
/**
```

- returns: In the Actions section: the number of actions this set will contain upon saving.

In the Accessories section: The number of accessories in the home.

```
*/
```

```
override func tableView(_ tableView: UITableView, numberOfRowsInSectionSection section: Int) -> Int {
```

```
    switch ActionSetTableViewSection(rawValue: section) {
```

```
        case .name?:
```

```
            return super.tableView(tableView, numberOfRowsInSectionSection: section)
```

```
        case .actions?:
```

```
            return max(actionSetCreator.allCharacteristics.count, 1)
```



```
case .accessories?:
```

```
    return displayedAccessories.count
```

```
case nil:
```

```
    fatalError("Unexpected ActionSetTableViewSection raw value.")
```

```
}
```

```
}
```

```
/**
```

Required override to allow for a tableView with both static and dynamic content.

Basically, since the superclass's `indentationLevelForRowAtIndexPath` is only expecting 1 row per section, just call the super class's implementation

for the first row.

```
*/
```

```
override func tableView(_ tableView: UITableView, indentationLevelForRowAtIndexPath: IndexPath:
```

```
IndexPath) -> Int {
```

```

        return super.tableView(tableView, indentationLevelForRowAt:
IndexPath(row: 0, section:
indexPath.section))

    }

    /// Removes the action associated with the index path.

    override func tableView(_ tableView: UITableView, commit editingStyle:
UITableViewCellEditingStyle, forRowAt indexPath: IndexPath) {
        if editingStyle == .delete {
            let characteristic = actionSetCreator.allCharacteristics[indexPath.row]
            actionSetCreator.removeTargetValueForCharacteristic(characteristic) {
                if self.actionSetCreator.containsActions {
                    tableView.deleteRows(at: [indexPath], with: .automatic)
                }
            }
        } else {
            tableView.reloadRows(at: [indexPath], with: .automatic)
        }
    }
}

```

```

    }

}

/// - returns: true for the Actions section; false otherwise.

override func tableView(_ tableView: UITableView, canEditRowAt indexPath:
IndexPath) -> Bool
{
    return ActionSetTableViewSection(rawValue: indexPath.section) == .actions
    &&
    home.isAdmin
}

/// - returns: UITableViewAutomaticDimension for dynamic sections, otherwise
the superclass's
implementation.

override func tableView(_ tableView: UITableView, heightForRowAt indexPath:
IndexPath) ->
CGFloat {

    switch ActionSetTableViewSection(rawValue: indexPath.section) {

```

```
case .name?:
```

```
return super.tableView(tableView, heightForRowAt: indexPath)
```

```
case .actions?, .accessories?:
```

```
return UITableView.automaticDimension
```

```
case nil:
```

```
fatalError("Unexpected ActionSetTableViewSection raw value.")
```

```
}
```

```
}
```

/// - returns: An action cell for the actions section, an accessory cell for the accessory section,

or the superclass's implementation.

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
```

```
UITableViewCell {
```

```
switch ActionSetTableViewSection(rawValue: indexPath.section) {
```

```

case .name?:

    return super.tableView(tableView, cellForRowAt: indexPath)

case .actions?:

    if actionSetCreator.containsActions {

        return self.tableView(tableView, actionCellForRowAtIndexPath:
indexPath)

    }

    else {

        return super.tableView(tableView, cellForRowAt: indexPath)

    }

case .accessories?:

    return self.tableView(tableView, accessoryCellForRowAtIndexPath:
indexPath)

case nil:

    fatalError("Unexpected ActionSetTableViewSection raw value.")

```

```

    }

}

// MARK: Helper Methods

/// Enables the save button if there is a valid name and at least one action.

private func enableSaveButtonIfNecessary() {

    saveButton.isEnabled = home.isAdmin && trimmedName.count > 0 &&
actionSetCreator.containsActions

}

/// - returns: The contents of the nameField, with whitespace trimmed from the
beginning and
end.

private var trimmedName: String {

    return nameField.text!.trimmingCharacters(in:
CharacterSet.whitespacesAndNewlines)

}

```



```
/**
```

- returns: true if there are no accessories in the home, we have no set action set,
or if our home no longer exists; false otherwise

```
*/
```

```
private func shouldPopViewController() -> Bool {
```

```
    if homeStore.home?.accessories.count == 0 && actionSet == nil {
```

```
        return true
```

```
    }
```

```
    return !homeStore.homeManager.homes.contains { $0 == homeStore.home }
```

```
}
```

/// - returns: An ActionCell instance with the target value for the characteristic at
the specified

index path.

```
private func tableView(_ tableView: UITableView,  
actionCellForRowAtIndexPath indexPath:
```

```
IndexPath) -> UITableViewCell {
```

```
    let cell = tableView.dequeueReusableCell(withIdentifier: Identifiers.actionCell,  
for: indexPath)
```

```
as! ActionCell
```

```
    let characteristic = actionSetCreator.allCharacteristics[indexPath.row] as  
HMCharacteristic
```

```
    if let target = actionSetCreator.targetValueForCharacteristic(characteristic) {
```

```
        cell.setCharacteristic(characteristic, targetValue: target)
```

```
    }
```

```
    return cell
```

```
}
```

```
/// - returns: An Accessory cell that contains an accessory's name.
```

```
private func tableView(_ tableView: UITableView,  
accessoryCellForRowAtIndexPath indexPath:
```

```
IndexPath) -> UITableViewCell {
```

```
/*
```

These cells are static, the identifiers are defined in the Storyboard, but they're not recognized here. In viewDidLoad:, we're registering UITableViewCell as the class for "AccessoryCell" and "UnreachableAccessoryCell".

We must configure these cells manually, the cells in the Storyboard are just for reference.

```
*/  
  
let accessory = displayedAccessories[indexPath.row]  
  
let cellIdentifier = accessory.isReachable ? Identifiers.accessoryCell :  
Identifiers.unreachableAccessoryCell  
  
let cell = tableView.dequeueReusableCell(withIdentifier: cellIdentifier, for:  
indexPath)  
  
cell.textLabel?.text = accessory.name  
  
if accessory.isReachable {  
  
    cell.textLabel?.textColor = UIColor.darkText
```

```

        cell.accessoryType = .disclosureIndicator

        cell.selectionStyle = .default

    }

    else {

        cell.textLabel?.textColor = UIColor.lightGray

        cell.accessoryType = .none

        cell.selectionStyle = .none

    }

    return cell

}

/// Shows the services in the selected accessory.

override func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {

    let cell = tableView.cellForRow(at: indexPath)!

    if cell.selectionStyle == .none {

        return
    }

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 642 ำงอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

    }

    if ActionSetTableViewSection(rawValue: indexPath.section) == .accessories {

        performSegue(withIdentifier: Identifiers.showServiceSegue, sender: cell)

    }

}

// MARK: HMHomeDelegate Methods

/**
    Pops the view controller if our configuration is invalid;
    reloads the view otherwise.

*/

func home(_ home: HMHome, didRemoveAccessory accessory: HMAccessory) {

    if shouldPopViewController() {

        self.dismiss(animated: true, completion: nil)

    }

    else {

```

```

        tableView.reloadData()

    }

}

/// Reloads the table view data.

func home(_ home: HMHome, didAddAccessory accessory: HMAccessory) {

    tableView.reloadData()

}

/// If our action set was removed, dismiss the view.

func home(_ home: HMHome, didRemoveActionSet actionSet: HMActionSet) {

    if actionSet == self.actionSet {

        self.dismiss(animated: true, completion: nil)

    }

}

}

}

}

ActionCell

import UIKit

```



```

import UIKit

/// A UITableViewCell subclass that displays a characteristic's 'target value'.

class ActionCell: UITableViewCell {

    /// Ignores the passed-in style and overrides it with .Subtitle.

    override init(style: UITableViewCell.CellStyle, reuseIdentifier: String?) {

        super.init(style: .subtitle, reuseIdentifier: reuseIdentifier)

        selectionStyle = .none
        detailTextLabel?.textColor = UIColor.lightGray
        accessoryType = .none
    }

    /// Required init.

    required init?(coder aDecoder: NSCoder) {

        super.init(coder: aDecoder)
    }
}

```

Sets the cell's text to represent a characteristic and target value.

For example, "Brightness → 60%"

Sets the subtitle to the service and accessory that this characteristic represents.

- parameter characteristic: The characteristic this cell represents.

- parameter targetValue: The target value from this action.

*/

```
func setCharacteristic(_ characteristic: HMCharacteristic, targetValue:
AnyObject) {
    let targetDescription = "\(characteristic.localizedDescription) →
    \(characteristic.localizedDescriptionForValue(targetValue))"
    titleLabel?.text = targetDescription

    let contextDescription = NSLocalizedString("%@ in %@", comment: "Service
in Accessory")

    if let service = characteristic.service, let accessory = service.accessory {

        detailTextLabel?.text = String(format: contextDescription, service.name,
        accessory.name)

    }
```

```

else {

    detailTextLabel?.text = NSLocalizedString("Unknown Characteristic",
comment: "Unknown
Characteristic")

}

}

}

}

ActionSetCreator
import HomeKit

/// A CharacteristicCellDelegate that builds an HMActionSet when it receives
delegate callbacks.

class ActionSetCreator: CharacteristicCellDelegate {

    // MARK: Properties

    var actionSet: HMActionSet?

    var home: HMHome

```

```
var saveError: Error?
```

```
/// The structure we're going to use to hold the target values.
```

```
let          targetValueMap          =          NSMapTable<HMCharacteristic,  
CellValueType>.strongToStrongObjects()
```

```
/// A dispatch group to wait for all of the individual components of the saving  
process.
```

```
let saveActionSetGroup = DispatchGroup()
```

```
required init(actionSet: HMActionSet?, home: HMHome) {
```

```
    self.actionSet = actionSet
```

```
    self.home = home
```

```
}
```

```
/**
```

If there is an action set, saves the action set and then updates its name.

Otherwise creates a new action set and adds all actions to it.

- parameter name: The new name for the action set.

- parameter completionHandler: A closure to call once the action set has been completely

saved.

```
*/  
  
func saveActionSetWithName(_ name: String, completionHandler: @escaping (_  
error: Error?)  
> Void) {  
    if let actionSet = actionSet {  
        saveActionSet(actionSet)  
        updateNameIfNecessary(name)  
    }  
  
    else {  
  
        createActionSetWithName(name)  
  
    }  
  
    saveActionSetGroup.notify(queue: DispatchQueue.main) {
```

```

        completionHandler(self.saveError)

        self.saveError = nil

    }

}

```

```
/**
```

Adds all of the actions that have been requested to the Action Set, then runs a completion block.

- parameter completion: A closure to be called when all of the actions have been added.

```
*/
```

```
func saveActionSet(_ actionSet: HMActionSet) {
```

```
    let actions = actionsFromMapTable(targetValueMap)
```

```
    for action in actions {
```

```
        saveActionSetGroup.enter()
```

```
        addAction(action, toActionSet: actionSet) { error in
```



```

        if let error = error {

            print("HomeKit: Error adding action: \(error.localizedDescription)")

            self.saveError = error

        }

        self.saveActionSetGroup.leave()

    }

}

}

/**
Sets the name of an existing action set.

- parameter name: The new name for the action set.

*/

func updateNameIfNecessary(_ name: String) {

    if actionSet?.name == name {

        return

    }

```

```

saveActionSetGroup.enter()

actionSet?.updateName(name) { error in

    if let error = error {

        print("HomeKit: Error updating name: \(error.localizedDescription)")

        self.saveError = error

    }

    self.saveActionSetGroup.leave()
}

/**
Creates and saves an action set with the provided name.

```

- parameter name: The name for the new action set.

```
*/
```

```
func createActionSetWithName(_ name: String) {
```

```
    saveActionSetGroup.enter()
```

```
    home.addActionSet(withName: name) { actionSet, error in
```

```

if let error = error {

    print("HomeKit: Error creating action set: \(error.localizedDescription)")

    self.saveError = error

}

else {

    // There is no error, so the action set has a value.

    self.saveActionSet(actionSet!)

}

self.saveActionSetGroup.leave()

}

}

/**

```

Checks to see if an action already exists to modify the same characteristic as the action passed in. If such an action exists, the method tells the existing action to update its target value. Otherwise, the new action is simply added to the action set.

- parameter action: The action to add or update.
- parameter actionSet: The action set to which to add the action.
- parameter completion: A closure to call when the addition has finished.

*/

```
func addAction(_ action: HMCharacteristicWriteAction<NSCopying>,
toActionSet actionSet:
```

```
HMAActionSet, completion: @escaping (Error?) -> Void) {
```

```
    if let existingAction = existingActionInActionSetMatchingAction(action) {
```

```
        existingAction.updateTargetValue(action.targetValue, completionHandler:
completion)
```

```
    }
```

```
    else {
```

```
        actionSet.addAction(action, completionHandler: completion)
```

```
    }
```

```
}
```

```
/**
```

Checks to see if there is already an HMCharacteristicWriteAction in

the action set that matches the provided action.

- parameter action: The action in question.

- returns: The existing action that matches the characteristic or nil if
there is no existing action.

*/

```
func existingActionInActionSetMatchingAction(_ action:
HMCharacteristicWriteAction<CellValueType>) ->
HMCharacteristicWriteAction<CellValueType>?
{
    if let actionSet = actionSet {
        for case let existingAction as
HMCharacteristicWriteAction<CellValueType> in
actionSet.actions {
            if action.characteristic == existingAction.characteristic {
                return existingAction
            }
        }
    }
}
```

```

    }

    }

    return nil

}

/**
Iterates over a map table of HMCharacteristic -> id objects and creates
an array of HMCharacteristicWriteActions based on those targets.
- parameter table: An NSMapTable mapping HMCharacteristics to id's.
- returns: An array of HMCharacteristicWriteActions.

*/

func actionsFromMapTable(_ table: NSMapTable<HMCharacteristic,
CellValueType>) ->

[HMCharacteristicWriteAction<CellValueType>] {

    return targetValueMap.keyEnumerator().allObjects.map { key in

        let characteristic = key as! HMCharacteristic

```



```

    let targetValue = targetValueMap.object(forKey: characteristic)!

    return HMCharacteristicWriteAction(characteristic: characteristic,
targetValue:
targetValue)

}

}

/**
- returns: true if the characteristic count is greater than zero;
false otherwise.
*/
var containsActions: Bool {

return !allCharacteristics.isEmpty

}

/**

All existing characteristics within HMCharacteristicWriteActions

and target values in the target value map.

```

```

*/

var allCharacteristics: [HMCharacteristic] {

    var characteristics = Set<HMCharacteristic>()

    if let actions = actionSet?.actions {

        let actionSetCharacteristics = actions.compactMap { action in

            return (action as?
HMCharacteristicWriteAction<CellValueType>)?.characteristic

        }

        characteristics.formUnion(actionSetCharacteristics)

    }

    characteristics.formUnion(targetValueMap.keyEnumerator().allObjects as!
[HMCharacteristic])

    return Array(characteristics)

}

```

/**

Searches through the target value map and existing
HMCharacteristicWriteActions

to find the target value for the characteristic in question.

- parameter characteristic: The characteristic in question.

- returns: The target value for this characteristic, or nil if there is no target.

*/

func targetValueForCharacteristic(_ characteristic: HMCharacteristic) ->
CellValueType? {

if let value = targetValueMap.object(forKey: characteristic) {

return value

}

else if let actions = actionSet?.actions {

for case let writeAction as HMCharacteristicWriteAction<CellValueType>
in actions {

if writeAction.characteristic == characteristic {

return writeAction.targetValue

```

    }

    }

}

```

```

return nil

```

```

}

```

```

/**

```

First removes the characteristic from the targetValueMap.

Then removes any HMCharacteristicWriteActions from the action set which set the specified characteristic.

- parameter characteristic: The HMCharacteristic to remove.

- parameter completion: The closure to invoke when the characteristic has been removed.

```

*/

```

```

func removeTargetValueForCharacteristic(_ characteristic: HMCharacteristic,
completion:

```

```
@escaping () -> Void) {
```

```
/*
```

We need to create a dispatch group here, because in many cases

there will be one characteristic saved in the Action Set, and one

in the target value map. We want to run the completion closure only one time,

to ensure we've removed both.

```
*/
```

```
let group = DispatchGroup()
```

```
if targetValueMap.object(forKey: characteristic) != nil {
```

```
// Remove the characteristic from the target value map.
```

```
DispatchQueue.main.async(group: group) {
```

```
self.targetValueMap.removeObject(forKey: characteristic)
```

```
}
```

```
}
```

```
if let actions = actionSet?.actions {
```

```
for case let action as HMCharacteristicWriteAction<CellValueType> in  
actions {
```

```
if action.characteristic == characteristic {
```

```
/*
```

Also remove the action, and only relinquish the dispatch group

once the action set has finished.

```
*/
```

```
group.enter()
```

```
actionSet?.removeAction(action) { error in
```

```
if let error = error {
```

```
    print(error.localizedDescription)
```

```
}
```

```
group.leave()
```

```
}
```

```
}
```

```
}
```

```
}
```

// Once we're positive both have finished, run the completion closure on the main queue.

```
group.notify(queue: DispatchQueue.main, execute: completion)
```

```
}
```


// MARK: Characteristic Cell Delegate

/**

Receives a callback from a CharacteristicCell with a value change.

Adds this value change into the targetValueMap, overwriting other value changes.

*/

```
func characteristicCell(_ cell: CharacteristicCell, didUpdateValue newValue:
CellValueType,
forCharacteristic characteristic: HMCharacteristic, immediate: Bool) {
    targetValueMap.setObject(newValue, forKey: characteristic)
}
```

/**

Receives a callback from a CharacteristicCell, requesting an initial value for a given characteristic.

It checks to see if we have an action in this Action Set that matches the characteristic.

If so, calls the completion closure with the target value.

*/

func characteristicCell(_ cell: CharacteristicCell,
readInitialValueForCharacteristic characteristic:

HMCharacteristic, completion: @escaping (CellValueType?, Error?) -> Void) {

if let value = targetValueForCharacteristic(characteristic) {

completion(value, nil)

return

}

characteristic.readValue { error in

/*

The user may have updated the cell value while the

read was happening. We check the map one more time.

*/

if let value = self.targetValueForCharacteristic(characteristic) {

```

        completion(value, nil)

    }

    else {

        completion(characteristic.value as? CellValueType, error)

    }

}

}

}

}

AddServicesViewController
import UIKit
import HomeKit

/**

```

A view controller that provides a list of services and lets the user select services to be added to

the provided Service Group.

The services are not added to the service group until the 'Done' button is pressed.

*/

```
class AddServicesViewController: HMCatalogViewController,
HMAccessoryDelegate {
```

```
// MARK: Types
```

```
struct Identifiers {
```

```
    static let serviceCell = "ServiceCell"
```

```
}
```

```
// MARK: Properties
```

```
lazy private var displayedAccessories = [HMAccessory]()
```

```
lazy private var displayedServicesForAccessory = [HMAccessory:
[HMSERVICE]]()
```

```
lazy private var selectedServices = [HMSERVICE]()
```

```
var serviceGroup: HMSERVICEGroup!
```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และส่ง 666 ไปถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```
// MARK: View Methods
```

```
/// Reloads internal data and view.
```

```
override func viewWillAppear(_ animated: Bool) {
```

```
    super.viewWillAppear(animated)
```

```
    selectedServices = []
```

```
    reloadTable()
```

```
}
```

```
/// Registers as the delegate for the home and all accessories.
```

```
override func registerAsDelegate() {
```

```
    super.registerAsDelegate()
```

```
    for accessory in homeStore.home!.accessories {
```

```
        accessory.delegate = self
```

```
    }
```

```
}
```

```
// MARK: Table View Methods
```

```

/// - returns: The number of displayed accessories.

override func numberOfSections(in tableView: UITableView) -> Int {

    return displayedAccessories.count

}

/// - returns: The number of displayed services for the provided accessory.

override func tableView(_ tableView: UITableView, numberOfRowsInSectionSection
section: Int) -> Int {

    let accessory = displayedAccessories[section]
    return displayedServicesForAccessory[accessory]!.count

}

/// - returns: A configured ServiceCell.

override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) ->

UITableViewCell {

    let cell = tableView.dequeueReusableCell(withIdentifier:
Identifiers.serviceCell, for:

```


indexPath) as! ServiceCell

let service = serviceAtIndexPath(indexPath)

cell.includeAccessoryText = false

cell.service = service

cell.accessoryType = selectedServices.contains(service) ? .checkmark : .none

return cell

}

/**

When an indexPath is selected, this function either adds or removes the selected service from

the

service group.

*/

```

override func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {

    // Get the service associated with this index.

    let service = serviceAtIndexPath(indexPath)

    // Call the appropriate add/remove operation with the closure from above.

    if let index = selectedServices.firstIndex(of: service) {
        selectedServices.remove(at: index)
    }
    else {
        selectedServices.append(service)
    }

    tableView.reloadRows(at: [indexPath], with: .automatic)

}

/// - returns: The name of the displayed accessory at the given section.

```

```
override func tableView(_ tableView: UITableView, titleForHeaderInSection: Int) ->
```

```
String? {
```

```
    return displayedAccessories[section].name
```

```
}
```

```
// MARK: Helper Methods
```

```
/**
```

Adds the selected services to the service group.

Calls the provided completion handler once all services have been added.

```
*/
```

```
func addSelectedServicesWithCompletionHandler(_ completion: @escaping () -> Void) {
```

```
    // Create a dispatch group for each of the service additions.
```

```
    let addServicesGroup = DispatchGroup()
```

```
    for service in selectedServices {
```

```

addServicesGroup.enter()

serviceGroup.addService(service) { error in

    if let error = error {

        self.displayError(error)

    }

    addServicesGroup.leave()

}

}

addServicesGroup.notify(queue: DispatchQueue.main, execute: completion)

}

/**

```

Finds the service at a specific index path.

- parameter indexPath: An NSIndexPath

- returns: The HMService at the given index path.

*/

```
private func serviceAtIndexPath(_ indexPath: IndexPath) -> HMService {

    let accessory = displayedAccessories[indexPath.section]

    let services = displayedServicesForAccessory[accessory]!

    return services[indexPath.row]

}
```

```
/**
```

Commits the changes to the service group
and dismisses the view.

```
*/
```

```
@IBAction func dismiss() {

    addSelectedServicesWithCompletionHandler {

        self.dismiss(animated: true, completion: nil)

    }

}
```

```
/// Resets internal data and view.
```

```
func reloadData() {
```

```
resetDisplayedServices()
```

```
tableView.reloadData()
```

```
}
```

```
/**
```

Updates internal array of accessories and the mapping

of accessories to selected services.

```
*/
```

```
func resetDisplayedServices() {
```

```
    displayedAccessories = []
```

```
    let allAccessories = home.accessories.sortByLocalizedNames()
```

```
    displayedServicesForAccessory = [:]
```

```
    for accessory in allAccessories {
```

```
        var displayedServices = [HMServices]()
```

```
        for service in accessory.services {
```

```
            if !serviceGroup.services.contains(service) && service.serviceType !=
```

```
HMServicesTypeAccessoryInformation {
```

```
                displayedServices.append(service)
```



```

    }

}

// Only add the accessory if it has displayed services.

if !displayedServices.isEmpty {

    displayedServicesForAccessory[accessory] =
displayedServices.sortByLocalizedNames()

    displayedAccessories.append(accessory)
}
}

}

// MARK: HMHomeDelegate Methods

/// Dismisses the view controller if our service group was removed.

func home(_ home: HMHome, didRemoveServiceGroup group:
HMServiceGroup) {

    if serviceGroup == group {

```

```

        self.dismiss(animated: true, completion: nil)

    }

}

/// Reloads the view if an accessory was added to HomeKit.

func home(_ home: HMHome, didAddAccessory accessory: HMAccessory) {

    reloadData()

    accessory.delegate = self
}

/// Dismisses the view controller if we no longer have accesories.

func home(_ home: HMHome, didRemoveAccessory accessory: HMAccessory) {

    if home.accessories.isEmpty {

        navigationController?.dismiss(animated: true, completion: nil)

    }

    reloadData()

}

```

```

// MARK: HMAccessoryDelegate Methods

// Accessory changes reload the data and view.

func accessory(_ accessory: HMAccessory, didUpdateNameFor service:
HMService) {

    reloadData()

}

func accessoryDidUpdateServices(_ accessory: HMAccessory) {

    reloadData()

}

}

ServiceGroupViewController

import UIKit

import HomeKit

```

```

/// A view controller that allows the user to add services to a service group.

class ServiceGroupViewController: HMCatalogViewController,
HMAccessoryDelegate {

    // MARK: Types

    struct Identifiers {

        static let serviceCell = "ServiceCell"

        static let addServicesSegue = "Add Services Plus"

    }

    // MARK: Properties

    @IBOutlet weak var plusButton: UIBarButtonItem!

    var serviceGroup: HMServiceGroup!

    lazy private var accessories = [HMAccessory]()

    lazy private var servicesForAccessory = [HMAccessory: [HMService]]()

```

```
// MARK: View Methods
```

```
/// Reloads the view.
```

```
override func viewWillAppear(_ animated: Bool) {
```

```
    super.viewWillAppear(animated)
```

```
    title = serviceGroup.name
```

```
    reloadData()
```

```
}
```

```
/// Pops the view controller if our data is invalid.
```

```
override func viewDidAppear(_ animated: Bool) {
```

```
    super.viewDidAppear(animated)
```

```
    if shouldPopViewController() {
```

```
        _ = navigationController?.popViewController(animated: true)
```

```
    }
```

```
}
```

```
// MARK: Table View Methods
```

/**

Generates the number of sections and adds a table view

back ground message, if required.

- returns: The number of accessories in the service group.

*/

override func numberOfSections(in tableView: UITableView) -> Int {

let sections = accessories.count

if sections == 0 {

setBackgroundMessage(NSLocalizedString("No Services", comment: "No Services"))

}

else {

setBackgroundMessage(nil)

}

return sections


```
}
```

/// - returns: The number of services for the accessory at the specified section.

```
override func tableView(_ tableView: UITableView, numberOfRowsInSectionSection: Int) -> Int {
```

```
    let accessory = accessories[section]
```

```
    let services = servicesForAccessory[accessory]
```

```
    return services?.count ?? 0
```

```
}
```

/// - returns: The name of the accessory at the specified section.

```
override func tableView(_ tableView: UITableView, titleForHeaderInSection: Int) ->
```

```
String? {
```

```
    return accessories[section].name
```

```
}
```

```

/// All cells in the table view represent services and can be deleted.

override func tableView(_ tableView: UITableView, canEditRowAt indexPath:
IndexPath) -> Bool
{

    return true

}

/// - returns: A ServiceCell with the service at the given index path.

override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) ->
UITableViewCell {

    let cell = tableView.dequeueReusableCell(withIdentifier:
Identifiers.serviceCell, for:
indexPath) as! ServiceCell

    let service = serviceAtIndex(indexPath)

    cell.includeAccessoryText = false

    cell.service = service

    return cell

}

```

```

/**

- returns: true if there are any services not already in the service group;

false otherwise.

*/

private func shouldEnableAdd() -> Bool {

    let unAddedServices =
home.servicesNotAlreadyInServiceGroup(serviceGroup)

    return unAddedServices.count != 0

}

/// Deleting a cell removes the corresponding service from the service group.

override func tableView(_ tableView: UITableView, commit editingStyle:
UITableViewCellEditingStyle, forRowAt indexPath: IndexPath) {

    if editingStyle == .delete {

        removeServiceAtIndexPath(indexPath)

    }

}

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 683 ึ่งอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

/**

Removes the service associated with the cell at a given index path.

- parameters indexPath: The NSIndexPath to remove.

*/

```
private func removeServiceAtIndexPath(_ indexPath: IndexPath) {
```

```
    let service = serviceAtIndexPath(indexPath)
```

```
    serviceGroup.removeService(service) { error in
```

```
        if let error = error {
```

```
            self.displayError(error)
```

```
        }
```

```
        self.reloadData()
```

```
    }
```

```
}
```

/**

Finds the service at a given index path.

- parameter indexPath: An NSIndexPath.

- returns: The service at the given index path

*/

```
private func serviceAtIndexPath(_ indexPath: IndexPath) -> HMService {  
  
    let accessory = accessories[indexPath.section]  
  
    let services = servicesForAccessory[accessory]!  
  
    return services[indexPath.row]  
  
}
```

/// Passes the service group into the AddServicesViewController

override func prepare(for segue: UIStoryboardSegue, sender: Any?) {

super.prepare(for: segue, sender: sender)

if segue.identifier == Identifiers.addServicesSegue {

let addServicesVC = segue.intendedDestinationViewController as!

AddServicesViewController

```

        addServicesVC.serviceGroup = serviceGroup

    }

}

```

// MARK: Helper Methods

/**

Resets accessory and service lists, resets the plus
button's enabled status and reloads the table view.

*/

```
private func reloadData() {
```

```
    resetLists()
```

```
    plusButton.isEnabled = shouldEnableAdd()
```

```
    tableView.reloadData()
```

```
}
```

/**

Resets the accessories array and the service-accessory mapping

using the original HomeKit objects.

*/

```
private func resetLists() {
```

```
    accessories = []
```

```
    servicesForAccessory = [:]
```

```
    for service in serviceGroup.services {
```

```
        if let accessory = service.accessory {
```

```
            if servicesForAccessory[accessory] == nil {
```

```
                accessories.append(accessory)
```

```
                servicesForAccessory[accessory] = [service]
```

```
            }
```

```
        else {
```

```
            servicesForAccessory[accessory]?.append(service)
```

```
        }
```

```
    }
```

```
}
```

```

// Sort all service lists.

for accessory in accessories {

    servicesForAccessory[accessory] =

servicesForAccessory[accessory]?.sortByLocalizedName()

}

// Sort accessory list.
accessories = accessories.sortByLocalizedName()
}

/**
- returns: true if our service group is not
in the home any more; false otherwise.

*/

private func shouldPopViewController() -> Bool {

    guard let home = homeStore.home else { return true }

    return !home.serviceGroups.contains { group in

```

```

        return group == serviceGroup

    }

}

/**
Registers as the delegate for the home and
all accessories which are related to our service group.
*/
override func registerAsDelegate() {
    super.registerAsDelegate()

    for service in serviceGroup.services {

        service.accessory?.delegate = self

    }

}

// MARK: HMHomeDelegate Methods

```

```

/// Pops the view controller if our service group has been deleted.

func home(_ home: HMHome, didRemoveServiceGroup group:
HMSERVICEGROUP) {

    if group == serviceGroup {

        _ = navigationController?.popViewController(animated: true)

    }

}

// Home and accessory changes result in a full data reload.

func home(_ home: HMHome, didAddService service: HMSERVICE,
toServiceGroup group:
HMSERVICEGROUP) {

    if serviceGroup == group {

        reloadData()

    }

}

```

```

func home(_ home: HMHome, didRemoveService service: HMService,
fromServiceGroup group:
HMSERVICEGROUP) {

    if serviceGroup == group {

        reloadData()

    }

}

func home(_ home: HMHome, didRemoveAccessory accessory: HMAccessory) {

    reloadData()

}

func accessoryDidUpdateServices(_ accessory: HMAccessory) {

    reloadData()

}

func accessory(_ accessory: HMAccessory, didUpdateNameFor service:
HMSERVICE) {

```

```
        reloadData()
```

```
    }
```

```
}
```

```
RoomPicker
```

```
import UIKit
```

```
import HomeKit
```

```
class RoomPicker: UITableViewController {
```

```
    /// The rooms to choose from.
```

```
    var rooms = [HMRoom]()
```

```
    /// The home holding the rooms.
```

```
    var home: HMHome? {
```

```
        didSet {
```

```
            rooms = []
```

```
            guard let home = home else { rooms = []; return }
```

```
            rooms = [home.roomForEntireHome()] + home.rooms
```



```

        selectedRow = roomRow

    }

}

/// The service which lives in one of the rooms.

var service: HMService? {

    didSet {

        selectedRow = roomRow

    }

}

/// The row of the selected room before any changes.

var roomRow: Int? {

    guard let room = service?.accessory?.room else { return nil }

    return rooms.firstIndex(of: room)

}

/// The row the user has selected, starting as the initial room row.

```

```

var selectedRow: Int?

/// Registers interest in home delegate callbacks.

override func viewDidLoad() {

    super.viewDidLoad()

    HomeStore.shared.addHomeDelegate(self)
}

/// Deregisters interest in home delegate callbacks.

deinit {

    HomeStore.shared.removeHomeDelegate(self)
}

/// Calls on the home store to save changes, if any.

override func viewWillDisappear(_ animated: Bool) {

    super.viewWillDisappear(animated)

    if let row = selectedRow,

```

```

let accessory = service?.accessory,

let home = home,

selectedRow != roomRow {

    HomeStore.shared.move(accessory, in: home, to: rooms[row])

}

}

/// Asks the user to supply a name for a new room, and then tries to create the room.
@IBAction func addRoom(_ sender: UIBarButtonItem) {

    let alert = UIAlertController(title: "Add a Room",

                                message: "Add a new room to the home.",

                                preferredStyle: .alert)

    alert.addTextField { $0.placeholder = "Name" }

    alert.addAction(UIAlertAction(title: "Create", style: .default) { _ in

        if let name = alert.textFields?[0].text {

            self.home?.addRoom(withName: name) { room, error in

                if let error = error {

```

```

        print("Error adding room: \(error)")

    } else if let home = self.home, let room = room {

        HomeStore.shared.home(home, didAdd: room)

    }

}

})

alert.addAction(UIAlertAction(title: "Cancel", style: .cancel))
present(alert, animated: true)
}

// MARK: - Table view data source

override func tableView(_ tableView: UITableView, numberOfRowsInSectionSection
section: Int) -> Int {

    return rooms.count

}

```

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) ->
```

```
UITableViewCell {
```

```
    let cell = tableView.dequeueReusableCell(withIdentifier: "RoomCell", for:
indexPath)
```

```
    // Fill in the name and check the selected row.
```

```
    cell.textLabel?.text = rooms[indexPath.row].name
```

```
    cell.accessoryType = indexPath.row == selectedRow ? .checkmark : .none
```

```
    return cell
```

```
}
```

```
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {
```

```
    // Turn all the check marks off.
```

```
    tableView.visibleCells.forEach { $0.accessoryType = .none }
```

```
    // Turn exactly one back on.
```

```

let cell = tableView.cellForRow(at: indexPath)

cell?.accessoryType = .checkmark

tableView.deselectRow(at: indexPath, animated: true)

// Remember which one was selected.

selectedRow = indexPath.row
}

// Enables the deletion of any room except the Default Room, which always
appears first.

override func tableView(_ tableView: UITableView, canEditRowAt indexPath:
IndexPath) -> Bool
{

return indexPath != IndexPath(row: 0, section: 0)

}

// Deletes a room.

override func tableView(_ tableView: UITableView, commit editingStyle:

```



```
UITableViewCellEditingStyle, forRowAt indexPath: IndexPath) {
```

```
    if editingStyle == .delete, let home = home {
```

```
        let room = rooms[indexPath.row]
```

```
        home.removeRoom(room) { error in
```

```
            if let error = error {
```

```
                print(error.localizedDescription)
```

```
            } else {
```

```
                HomeStore.shared.home(home, didRemove: room)
```

```
            }
```

```
        }
```

```
    }
```

```
}
```

```
}
```

```
NameEditor
```

```
import UIKit
```

```
import HomeKit
```

```
class NameEditor: UITableViewController {
```

```
@IBOutlet weak var nameField: UITextField!
```

```
var service: HMService?
```

```
override func viewDidLoad() {
```

```
    super.viewDidLoad()
```

```
    nameField.text = service?.name
```

```
}
```

```
override func viewWillDisappear(_ animated: Bool) {
```

```
    super.viewWillDisappear(animated)
```

```
    if let name = nameField.text,
```

```
        let service = service {
```

```
            HomeStore.shared.updateService(service, name: name)
```

```
        }
```

```
    }
```

```
}
```

```
RoomPicker+HomeDelegate
```

```
import HomeKit
```

```
/// Handle the home delegate callbacks.
```

```
extension RoomPicker: HMHomeDelegate {
```

```
    func home(_ home: HMHome, didUpdate room: HMRoom, for accessory: HMAccessory) {
```

```
        guard home == self.home, accessory == service?.accessory else { return }
```

```
        // Record the row of the newly selected room.
```

```
        selectedRow = roomRow
```

```
        // Reset the checkmarks
```

```
        tableView.indexPathsForVisibleRows?.forEach { indexPath in
```

```
            let cell = tableView.cellForRow(at: indexPath)
```

```
            cell?.accessoryType = indexPath.row == selectedRow ? .checkmark : .none
```

```
        }
```

```
}
```

```
func home(_ home: HMHome, didAdd room: HMRoom) {
```

```
    guard home == self.home else { return }
```

```
    // Add a row at the end.
```

```
    let indexPaths = [IndexPath(row: rooms.count, section: 0)]
```

```
    rooms.append(room)
```

```
    tableView.insertRows(at: indexPaths, with: .fade)
```

```
}
```

```
func home(_ home: HMHome, didUpdateNameFor room: HMRoom) {
```

```
    guard home == self.home else { return }
```

```
    // Rewrite all the visible room names.
```

```
    tableView.indexPathsForVisibleRows?.forEach { indexPath in
```

```
        let cell = tableView.cellForRow(at: indexPath)
```

```
        cell?.textLabel?.text = rooms[indexPath.row].name
```

เอกสารนี้เป็นเอกสารสงวนลิขสิทธิ์สำหรับการใช้งานเพื่อการศึกษาเท่านั้น เมื่ออนุญาตให้นำไปใช้ประโยชน์ด้านการค้า

ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

    }

}

func home(_ home: HMHome, didRemove room: HMRoom) {

    guard home == self.home else { return }

    // Delete the appropriate row.

    if let row = rooms.firstIndex(of: room) {
        let indexPaths = [IndexPath(row: row, section: 0)]
        rooms.remove(at: row)
        tableView.deleteRows(at: indexPaths, with: .fade)
    }
}

}

}

```

AccessoryList+AccessoryDelegate

```
import HomeKit
```

```
extension AccessoryList: HMAccessoryDelegate {
```

```

/// Handles characteristic value updates.

func accessory(_ accessory: HMAccessory, service: HMSERVICE,
didUpdateValueFor
characteristic: HMCharacteristic) {

    if let item = kilgoServices.firstIndex(of: service) {

        let cell = collectionView.cellForItem(at: IndexPath(item: item, section: 0))
as? DeviceCell

        cell?.redrawState()

    }
}

func accessory(_ accessory: HMAccessory, didUpdateNameFor service:
HMSERVICE) {

    if let item = kilgoServices.firstIndex(of: service) {

        let cell = collectionView.cellForItem(at: IndexPath(item: item, section: 0))
as? DeviceCell

        cell?.nameLabel.text = service.name

    }

}
}

```



```
}
```

AccessoryList+HomeDelegate

```
import HomeKit
```

```
/// Handle the home delegate callbacks.
```

```
extension AccessoryList: HMHomeDelegate {
```

```
func homeDidUpdateName(_ home: HMHome) {
```

```
    guard home == self.home else { return }
```

```
    title = home.name
```

```
}
```

```
func home(_ home: HMHome, didAdd accessory: HMAccessory) {
```

```
    guard home == self.home else { return }
```

```
    // Make sure the new accessory generates callbacks to the home store.
```

```
    accessory.delegate = HomeStore.shared
```

```

reloadData()

}

func home(_ home: HMHome, didUpdate room: HMRoom, for accessory:
HMAccessory) {

    for service in accessory.services {

        if let item = kilgoServices.firstIndex(of: service) {

            let cell = collectionView.cellForItem(at: IndexPath(item: item, section: 0))
as? DeviceCell

            cell?.roomLabel.text = room.name

        }

    }

}

}

}

func home(_ home: HMHome, didRemove accessory: HMAccessory) {

    guard home == self.home else { return }

```

```

navigationController?.popToRootViewController(animated: true)

reloadData()

}

func home(_ home: HMHome, didUpdateNameFor room: HMRoom) {

    for cell in collectionView.visibleCells {

        (cell as? DeviceCell)?.roomLabel.text = room.name

    }

}

func home(_ home: HMHome, didEncounterError error: Error, for accessory:
HMAccessory) {

    print(error.localizedDescription)

}

}

AccessoryList+HomeManagerDelegate

import HomeKit

```

```

/// Handle the home manager delegate callbacks.

extension AccessoryList: HMHomeManagerDelegate {

    func homeManagerDidUpdateHomes(_ manager: HMHomeManager) {

        setOrAddHome(manager: manager)

    }

    func homeManagerDidUpdatePrimaryHome(_ manager: HMHomeManager) {

        setOrAddHome(manager: manager)

    }

    /// Sets the home to either the primary home, or the first home, or a home that the
    user creates.

    func setOrAddHome(manager: HMHomeManager) {

        if manager.primaryHome != nil {

            home = manager.primaryHome

        } else if let firstHome = manager.homes.first {

            home = firstHome

        } else {

```

```

let alert = UIAlertController(title: "Add a Home",

                                message: "There aren't any homes in the database. Create
a home to work
with.",

                                preferredStyle: .alert)

alert.addTextField { $0.placeholder = "Name" }

alert.addAction(UIAlertAction(title: "Create", style: .default) { _ in

    if let name = alert.textFields?[0].text {

        manager.addHome(withName: name) { home, error in

            if let error = error {

                print("Error adding home: \(error)")

            } else {

                self.home = home

            }

        }

    }

})

present(alert, animated: true)

```

```

    }

}

}

```

```
AccessoryList
```

```
import UIKit
```

```
import HomeKit
```

```
/// - Tag: AccessoryList
```

```
class AccessoryList: UIViewController {
```

```
    /// The filtered list of services that the app displays.
```



```
var kilgoServices = [HMService]() // These are called "accessories" in the UI.
```

```
/// The home whose accessories the app displays.
```

```
var home: HMHome? {
```

```
    didSet {
```

```
        home?.delegate = HomeStore.shared
```

```
        reloadData()
```

```
    }
```

```
}
```

```
/// Resets the list of Kilgo services from the currently set home.
```

```
func reloadData() {
```

```
    kilgoServices = []
```

```
    guard let home = home else { return }
```

```
for accessory in home.accessories.filter({ $0.manufacturer == "Kilgo Devices,  
Inc." }) {
```

```
    accessory.delegate = HomeStore.shared
```

```
    for service in accessory.services.filter({ $0.isUserInteractive }) {
```

```
        kilgoServices.append(service)
```

```
        // Ask for notifications from any characteristics that support them.
```

```
        for characteristic in service.characteristics.filter({
```

```
$0.properties.contains(HMCharacteristicPropertySupportsEventNotification)
```

```
}) {
```

```
characteristic.enableNotification(true) { _ in }
```

```
}
```

```
}
```

```
}
```

```
collectionView.reloadData()
```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```
}
```

```
/// Registers this view controller to receive various delegate callbacks.
```

```
override func viewDidLoad() {
```

```
    navigationItem.title = "My Farm"
```

```
    if #available(iOS 13.0, *) {
```

```
        let navBarAppearance = UINavigationBarAppearance()
```

```
        navBarAppearance.configureWithOpaqueBackground()
```

```
        navBarAppearance.titleTextAttributes = [.foregroundColor: UIColor.black]
```

```
        navBarAppearance.largeTitleTextAttributes = [.foregroundColor: UIColor.black]
```

```
        navBarAppearance.backgroundColor = UIColor.init(hexString: "F2EEE5")
```

```
        navigationController?.navigationBar.standardAppearance = navBarAppearance
```

```
        navigationController?.navigationBar.scrollEdgeAppearance = navBarAppearance
```

```
}
```

```
super.viewDidLoad()
```

```
HomeStore.shared.homeManager.delegate = self
```

```
HomeStore.shared.addHomeDelegate(self)
```

```
HomeStore.shared.addAccessoryDelegate(self)
```

```
}
```

```
/// Deregisters this view controller as various kinds of delegate.
```

```
deinit {
```



```
HomeStore.shared.homeManager.delegate = nil
```

```
HomeStore.shared.removeHomeDelegate(self)
```

```
HomeStore.shared.removeAccessoryDelegate(self)
```

```
}
```

```
/// Prepares to show the detail view for an accessory.
```

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
```

```
if segue.identifier == "showDetail",
```

```
let accessoryDetail = segue.destination as? AccessoryDetail,
```

```
let button = sender as? UIButton {
```

```
    accessoryDetail.service = kilgoServices[button.tag]
```

```
    accessoryDetail.home = home
```

```
}
```

```
}
```

```
/// Starts the add accessory flow.
```

```
@IBAction func tapAdd(_ sender: UIBarButtonItem) {
```

```
    home?.addAndSetupAccessories(completionHandler: { error in
```

```
        if let error = error {
```

```
            print(error)
```

```
        } else {
```

```
            // Make no assumption about changes; just reload everything.
```

```
            self.reloadData()
```

```
        }
```

```
})
```

```
}
```

```
// MARK: UICollectionViewDataSource
```

```
/// Tells the collection view to create one cell for each service.
```

```
override func collectionView(_ collectionView: UICollectionView,  
numberOfItemsInSection
```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```
section: Int) -> Int {
```

```
    return kilgoServices.count
```

```
}
```

```
/// Supplies the collection view with configured cells.
```

```
override func collectionView(_ collectionView: UICollectionView,  
cellForItemAt indexPath:
```

```
IndexPath) -> UICollectionViewCell {
```

```
    let cell = collectionView.dequeueReusableCell(withReuseIdentifier:  
"DeviceCell", for:
```

```
indexPath)
```

```
cell.layer.cornerRadius = 10
```

```
if let deviceCell = cell as? DeviceCell {
```

```
    deviceCell.service = kilgoServices[indexPath.item]
```

```
    deviceCell.disclosureButton.tag = indexPath.item
```

```
}
```

```
return cell
```



```
}
```

```
/// Passes taps along to the cell.
```

```
override func collectionView(_ collectionView: UICollectionView,  
didSelectItemAt indexPath:  
IndexPath) {
```

```
(collectionView.cellForItem(at: indexPath) as? DeviceCell)?.tap()
```

```
}
```

```
}
```

```
DeviceCell
```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```
import UIKit

import HomeKit

class DeviceCell: UICollectionViewCell {

    @IBOutlet weak var imageView: UIImageView!

    @IBOutlet weak var roomLabel: UILabel!

    @IBOutlet weak var nameLabel: UILabel!

    @IBOutlet weak var stateLabel: UILabel!

    @IBOutlet weak var disclosureButton: UIButton!

    /// The service that this cell represents.

    var service: HMService? {

        didSet {

            imageView.image = service?.icon

            roomLabel.text = service?.accessory?.room?.name

            nameLabel.text = service?.name
```

```

stateLabel.text = "Updating..."

        readAndRedraw(characteristic: service?.primaryDisplayCharacteristic,
animated: true)

    }

}

/// Reads the characteristic value from the HomeKit database, and updates the UI.
func readAndRedraw(characteristic: HMCharacteristic?, animated: Bool) {
    guard
        let characteristic = characteristic,
        characteristic.properties.contains(HMCharacteristicPropertyReadable),
        let accessory = characteristic.service?.accessory,
        accessory.isReachable else {

        stateLabel.text = "Unreachable"

        return

    }
}

```

```
characteristic.readValue { error in  
  
    self.redrawState(error: error)  
  
}  
  
}
```

```
/// Updates the UI to reflect the given state.
```

```
func redrawState(error: Error? = nil) {
```

```
    imageView.image = service?.icon
```

```
    if let error = error {
```

```
        print(error.localizedDescription)
```

```
        stateLabel.text = "Update error!"
```

```
    } else {
```

```
        stateLabel.text = service?.state
```

```
    }
```

```
}
```

```
/// Informs the cell that it's been tapped.
```

```

func tap() {

    if let characteristic = service?.primaryControlCharacteristic,

        let value = characteristic.value as? Bool {

        // Provide visual feedback that the item was tapped.

        bounce()

        // Write the new value to HomeKit.
        characteristic.writeValue(!value) { error in
            self.redrawState(error: error)
        }
    }
}

```

```

/// Animates the cell size in a way that looks like a little bounce.

```

```

private func bounce() {

```

```

    UIView.animate(withDuration: 0.05, animations: {

```

```

        self.transform = CGAffineTransform(scaleX: 0.95, y: 0.95)
    })
}

```

```
}) {_ in
```

```
UIView.animate(withDuration: 0.10, animations: {
```

```
self.transform = CGAffineTransform(scaleX: 1.05, y: 1.05)
```

```
}) {_ in
```

```
UIView.animate(withDuration: 0.15, animations: {
```

```
self.transform = .identity
```

```
})
```

```
}
```

```
}
```

```
}
```

```
}
```

```
HomeStore+AccessoryDelegate
```

```
import HomeKit
```

```
extension HomeStore {
```

```
/// Registers an object as an accessory delegate.
```

```
func addAccessoryDelegate(_ delegate: NSObject) {
```

```
accessoryDelegates.insert(delegate)
```



```

}

/// Deregisters a particular accessory delegate.

func removeAccessoryDelegate(_ delegate: NSObject) {

    accessoryDelegates.remove(delegate)

}

/// Deregisters all accessory delegates.
func removeAllAccessoryDelegates() {
    accessoryDelegates.removeAll()
}

}

extension HomeStore: HMAccessoryDelegate {

    // The home store's only interest in the accessory updates is distributing them

    // to the objects that have registered as accessory delegates. Each of these

    // methods therefore simply passes along the call to all the items in the set,

```

```

// after first ensuring that the item is in fact an accessory delegate.

func accessory(_ accessory: HMAccessory, didUpdateNameFor service:
HMService) {

    accessoryDelegates.forEach {

        guard let delegate = $0 as? HMAccessoryDelegate else { return }

        delegate.accessory?(accessory, didUpdateNameFor: service)

    }

}

func accessoryDidUpdateReachability(_ accessory: HMAccessory) {

    accessoryDelegates.forEach {

        guard let delegate = $0 as? HMAccessoryDelegate else { return }

        delegate.accessoryDidUpdateReachability?(accessory)

    }

}

func accessoryDidUpdateServices(_ accessory: HMAccessory) {

```

```

accessoryDelegates.forEach {

    guard let delegate = $0 as? HMAccessoryDelegate else { return }

    delegate.accessoryDidUpdateServices?(accessory)

}

}

func accessory(_ accessory: HMAccessory, service: HMAccessoryService,
didUpdateValueForCharacteristic: HMCharacteristic) {

    accessoryDelegates.forEach {

        guard let delegate = $0 as? HMAccessoryDelegate else { return }

        delegate.accessory?(accessory, service: service, didUpdateValueFor:
characteristic)

    }

}

}

}

KilgoService

import HomeKit

```

```
extension HMService {
```

```
    /// The service types that Kilgo garage doors support.
```

```
    enum KilgoServiceType {
```

```
        case lightBulb, garageDoor, unknown
```

```
    }
```

```
    /// The Kilgo service type for this service.
```

```
    var kilgoServiceType: KilgoServiceType {
```

```
        switch serviceType {
```

```
            case HMServiceTypeLightbulb: return .lightBulb
```

```
            case HMServiceTypeGarageDoorOpener: return .garageDoor
```

```
            default: return .unknown
```

```
        }
```

```
    }
```

```
    /// The primary characteristic type to be controlled, given the service type.
```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น เมื่อเผยแพร่ให้นำไปใช้ประโยชน์ด้านการค้า

ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

var primaryControlCharacteristicType: String? {

    switch kilgoServiceType {

        case .lightBulb: return HMCharacteristicTypePowerState

        case .garageDoor: return HMCharacteristicTypeTargetDoorState

        case .unknown: return nil

    }

}

/// The primary characteristic controlled by tapping the accessory cell in the
accessory list.

var primaryControlCharacteristic: HMCharacteristic? {

    return characteristics.first { $0.characteristicType ==
primaryControlCharacteristicType }

}

/// The primary characteristic type to be displayed, given the service type.

var primaryDisplayCharacteristicType: String? {

    switch kilgoServiceType {

```

```

case .lightBulb: return HMCharacteristicTypePowerState

case .garageDoor: return HMCharacteristicTypeCurrentDoorState

case .unknown: return nil

}

}

/// The primary characteristic visible in the accessory cell in the accessory list.
var primaryDisplayCharacteristic: HMCharacteristic? {
    return characteristics.first { $0.characteristicType ==
primaryDisplayCharacteristicType }
}

/// The custom displayable characteristic types specific to Kilgo devices.
enum KilgoCharacteristicTypes: String {

    case fadeRate = "7E536242-341C-4862-BE90-272CE15BD633"

}

/// The list of characteristics to display in the UI.

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้


```

var displayableCharacteristics: [HMCharacteristic] {

    let characteristicTypes = [HMCharacteristicTypePowerState,

                                HMCharacteristicTypeBrightness,

                                HMCharacteristicTypeHue,

                                HMCharacteristicTypeSaturation,

                                HMCharacteristicTypeTargetDoorState,

                                HMCharacteristicTypeCurrentDoorState,

                                HMCharacteristicTypeObstructionDetected,

                                HMCharacteristicTypeTargetLockMechanismState,

                                HMCharacteristicTypeCurrentLockMechanismState,

                                KilgoCharacteristicTypes.fadeRate.rawValue]

    return characteristicTypes.filter {
        characteristicTypes.contains($0.characteristicType) }

}

```

/// A graphical representation of the service, given its current state.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 735 จึงอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```
var icon: UIImage? {
```

```
    let (_, icon) = stateAndIcon
```

```
    return icon
```

```
}
```

```
/// A textual representation of the current state of the service.
```

```
var state: String {
```

```
    let (state, _) = stateAndIcon
```

```
    return state
```

```
}
```

```
/// A tuple containing a string and icon representing the current state of the service.
```

```
private var stateAndIcon: (String, UIImage?) {
```

```
    switch kilgoServiceType {
```

```
        case .garageDoor:
```

```
            if let value = primaryDisplayCharacteristic?.value as? Int,
```

```
                let doorState = HMCharacteristicValueDoorState(rawValue: value) {
```

```
                    switch doorState {
```

```

        case .open: return ("Open", #imageLiteral(resourceName: "door-open"))

        case .closed: return ("Closed", #imageLiteral(resourceName: "door-
closed"))

        case .opening: return ("Opening", #imageLiteral(resourceName: "door-
opening"))

        case .closing: return ("Closing", #imageLiteral(resourceName: "door-
closing"))

        case .stopped: return ("Stopped", #imageLiteral(resourceName: "door-
closed"))

        @unknown default: return ("Unknown", nil)
    }
} else {
    return ("Unknown", #imageLiteral(resourceName: "door-closed"))
}

case .lightBulb:

    if let value = primaryDisplayCharacteristic?.value as? Bool {

        if value {

            return ("On", #imageLiteral(resourceName: "bulb-on"))

        } else {

```

```

        return ("Off", #imageLiteral(resourceName: "bulb-off"))

    }

    } else {

        return ("Unknown", #imageLiteral(resourceName: "bulb-off"))

    }

case .unknown:

    return ("Unknown", nil)

}

}

}

HomeStore

import UIKit

import HomeKit

class HomeStore: NSObject, HMHomeManagerDelegate {

    static let shared = HomeStore()

    // MARK: Properties

```

```

/// The current 'selected' home.

var home: HMHome?

var homeDelegates = Set<NSObject>()

var accessoryDelegates = Set<NSObject>()

/// The singleton home manager.
var homeManager = HMHomeManager()
}

extension HomeStore {

/// Updates the name of a service and informs all accessory delegates.

func updateService(_ service: HMService, name: String) {

    service.updateName(name) { error in

        if let error = error {

            print(error)

```

```

    } else if let accessory = service.accessory {

        self.accessory(accessory, didUpdateNameFor: service)

    }

}

```

/// Moves an accessory to a given room and informs all the home delegates.

```

func move(_ accessory: HMAccessory, in home: HMHome, to room: HMRoom)
{
    home.assignAccessory(accessory, to: room) { error in
        if let error = error {
            print(error)
        } else {
            self.home(home, didUpdate: room, for: accessory)
        }
    }
}
}

```



```

/// Removes an accessory from a home and informs all the home delegates.

func remove(_ accessory: HMAccessory, from home: HMHome) {

    home.removeAccessory(accessory) { error in

        if let error = error {

            print(error)

        } else {

            self.home(home, didRemove: accessory)

        }

    }

}

HMCatalogViewController

import UIKit

import HomeKit

/**

```

The super class for most table view controllers in this app. It manages home

delegate registration and facilitates 'popping back' when it's discovered that

a home has been deleted.

*/

```
class HMCatalogViewController: UITableViewController, HMHomeDelegate {
```

```
// MARK: Properties
```

```
var homeStore: HomeStore {
```

```
    return HomeStore.shared
```

```
}
```

```
var home: HMHome! {
```

```
    return homeStore.home
```

```
}
```

```
// MARK: View Methods
```

```
/**
```

Evaluates whether or not the view controller should pop to

the list of homes.

- returns: true if this instance is not the root view controller

and the home is nil; false otherwise.

*/

```
private func shouldPopViewController() -> Bool {
```

```
    if let rootViewController = navigationController?.viewControllers.first
```

```
        , rootViewController == self {
```

```
        return false
```

```
    }
```

```
    return home == nil
```

```
}
```

/// Pops the view controller, if required. Invokes the delegate registration method.

```
override func viewWillAppear(_ animated: Bool) {
```

```
    super.viewWillAppear(animated)
```

```
    if shouldPopViewController() {
```

```
// Pop to root view controller if our home was destroyed while we were away.
```

```
_ = navigationController?.popToRootViewController(animated: true)
```

```
return
```

```
}
```

```
registerAsDelegate()
```

```
}
```

```
// MARK: Delegate Registration
```

```
/**
```

A hierarchical method, to be overridden by superclasses.

The base implementation registers as the delegate for the HomeStore's home.

Thus, any subclasses may override this, register as the delegate for any

objects they please, and then call `super.registerAsDelegate()` to register

as the home delegate as well.

This method will be called when the view appears.

เอกสารนี้เป็นเอกสารที่สงวนลิขสิทธิ์สำหรับการใช้งานเพื่อการศึกษาเท่านั้น เมื่ออนุญาตให้นำไปใช้ประโยชน์ด้านการค้า

ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 744 ึ่งอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

*/

func registerAsDelegate() {

    homeStore.home?.delegate = self

}

}

HomeStore+HomeDelegate

import HomeKit

extension HomeStore {

    /// Registers an object as a home delegate.

    func addHomeDelegate(_ delegate: NSObject) {

        homeDelegates.insert(delegate)

    }

    /// Deregisters a particular home delegate.

    func removeHomeDelegate(_ delegate: NSObject) {

        homeDelegates.remove(delegate)

    }

```

```

/// Deregisters all home delegates.

func removeAllHomeDelegates() {

    homeDelegates.removeAll()

}

}

extension HomeStore: HMHomeDelegate {

    // The home store's only interest in the home updates is distributing them
    // to the objects that have registered as home delegates. Each of these
    // methods therefore simply passes along the call to all the items in the set,
    // after first ensuring that the item is in fact a home delegate.

    func homeDidUpdateName(_ home: HMHome) {

        homeDelegates.forEach {

            guard let delegate = $0 as? HMHomeDelegate else { return }

            delegate.homeDidUpdateName?(home)

```



```

    }

}

func home(_ home: HMHome, didAdd accessory: HMAccessory) {

    homeDelegates.forEach {

        guard let delegate = $0 as? HMHomeDelegate else { return }

        delegate.home?(home, didAdd: accessory)

    }

}

func home(_ home: HMHome, didUpdate room: HMRoom, for accessory:
HMAccessory) {

    homeDelegates.forEach {

        guard let delegate = $0 as? HMHomeDelegate else { return }

        delegate.home?(home, didUpdate: room, for: accessory)

    }

}

```

```

func home(_ home: HMHome, didRemove accessory: HMAccessory) {

    homeDelegates.forEach {

        guard let delegate = $0 as? HMHomeDelegate else { return }

        delegate.home?(home, didRemove: accessory)

    }

}

```

```

func home(_ home: HMHome, didAdd room: HMRoom) {

    homeDelegates.forEach {

        guard let delegate = $0 as? HMHomeDelegate else { return }

        delegate.home?(home, didAdd: room)

    }

}

```

```

func home(_ home: HMHome, didUpdateNameFor room: HMRoom) {

    homeDelegates.forEach {

        guard let delegate = $0 as? HMHomeDelegate else { return }

        delegate.home?(home, didUpdateNameFor: room)

```

```

    }

}

func home(_ home: HMHome, didRemove room: HMRoom) {

    homeDelegates.forEach {

        guard let delegate = $0 as? HMHomeDelegate else { return }

        delegate.home?(home, didRemove: room)

    }

}

func home(_ home: HMHome, didEncounterError error: Error, for accessory:
HMAccessory) {

    homeDelegates.forEach {

        guard let delegate = $0 as? HMHomeDelegate else { return }

        delegate.home?(home, didEncounterError: error, for: accessory)

    }

}

}

```

KilgoCharacteristic

import HomeKit

extension HMCharacteristic {

var stateNames: [String] {

switch characteristicType {

case HMCharacteristicTypeCurrentLockMechanismState: return ["Unsecured",
"Secured",

"Jammed", "Unknown"]

case HMCharacteristicTypeTargetLockMechanismState: return ["Not Locked",
"Locked"]

case HMCharacteristicTypeCurrentDoorState: return ["Open", "Closed",
"Opening", "Closing",

"Stopped"]

case HMCharacteristicTypeTargetDoorState: return ["Open", "Closed"]

default: return []

}

}

```
/// Indicates if you can write to the characteristic.
```

```
var isWriteable: Bool {
```

```
    return properties.contains(HMCharacteristicPropertyWritable)
```

```
}
```

```
/// Indicates if you can read from the characteristic.
```

```
var isReadable: Bool {
```

```
    return properties.contains(HMCharacteristicPropertyReadable)
```

```
}
```

```
/// Indicates if the characteristic is both readable and writable.
```

```
var isReadWrite: Bool {
```

```
    return isReadable && isWriteable
```

```
}
```

```
/// Indicates that the characteristic value is a floating point number.
```

```
var isFloat: Bool {
```

```
    return metadata?.format == HMCharacteristicMetadataFormatFloat
```

เอกสารนี้เป็นเอกสารที่สงวนลิขสิทธิ์ไว้สำหรับใช้ในงานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า

ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และ 751 ไปถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```
}
```

```
/// Indicates that the characteristic value is a signed integer.
```

```
var isInt: Bool {
```

```
    return metadata?.format == HMCharacteristicMetadataFormatInt
```

```
}
```

```
/// Indicates that the characteristic value is an unsigned integer.
```

```
var isUInt: Bool {
```

```
    return metadata?.format == HMCharacteristicMetadataFormatUInt8
```

```
    || metadata?.format == HMCharacteristicMetadataFormatUInt16
```

```
    || metadata?.format == HMCharacteristicMetadataFormatUInt32
```

```
    || metadata?.format == HMCharacteristicMetadataFormatUInt64
```

```
}
```

```
/// Indicates that the characteristic value is a number.
```

```
// var isNumeric: Bool {
```



```

//      return isInt || isFloat || isUInt

//  }

/// Indicates that the characteristic value is a Boolean.

var isBool: Bool {

    return metadata?.format == HMCharacteristicMetadataFormatBool

}

/// The difference between the characteristic's maximum and minimum values.

var span: Decimal {

    guard let max = metadata?.maximumValue?.decimalValue,

        let min = metadata?.minimumValue?.decimalValue else {

        return 1.0

    }

    return max - min

}

}

/// The characteristic value as a float.

```

```

var floatValue: Float? {

    return (value as? NSNumber)?.floatValue

}

/// The characteristic value as a decimal value.

var decimalValue: Decimal? {

    return (value as? NSNumber)?.decimalValue

}

/// Returns the value that you get by mapping the given fraction
/// to the characteristic's span, accounting for step size.

func valueFor(fraction: Float) -> Decimal {

    let interval = metadata?.stepValue?.decimalValue ?? 1.0

    var inVal = (Decimal(Double(fraction)) * span) / interval

    var outVal: Decimal = 0.0

    NSDecimalRound(&outVal, &inVal, 0, .plain)

    return outVal * interval

}

```

```

/// A name that best represents the characteristic in the UI.

var displayName: String {

    return metadata?.manufacturerDescription ?? localizedDescription

}

/// A string that represents the characteristic's value.

var formattedValueString: String {

    guard

    let value = value,

    let metadata = metadata else { return "—" }

    // Use the metadata to drive the string formatting.

    switch metadata.format {

    case HMCharacteristicMetadataFormatString:

        return value as? String ?? "—"

    case HMCharacteristicMetadataFormatInt,

```

HMCharacteristicMetadataFormatUInt8,

HMCharacteristicMetadataFormatUInt16,

HMCharacteristicMetadataFormatUInt32,

HMCharacteristicMetadataFormatUInt64:

guard let intValue = value as? Int else { return "—" }

guard stateNames.isEmpty else { return stateNames[intValue] }

return String(intValue)

case HMCharacteristicMetadataFormatFloat:

guard let floatValue = floatValue else { return "—" }

return String(format: "%.1f", floatValue)

case HMCharacteristicMetadataFormatBool:

guard let boolValue = value as? Bool else { return "—" }

return boolValue ? "YES" : "NO"

default:

return "—"

```
}
```

```
}
```

```
}
```

```
ATCTextField
```

```
import UIKit
```

```
import UIKit
```

```
class ATCTextField: UITextField {
```

```
    let padding = UIEdgeInsets(top: 0, left: 15, bottom: 0, right: 5)
```

```
    override open func textRect(forBounds bounds: CGRect) -> CGRect {
```

```
        return bounds.inset(by: padding)
```

```
    }
```

```
    override open func placeholderRect(forBounds bounds: CGRect) -> CGRect {
```

```
        return bounds.inset(by: padding)
```

```

    }

    override open func editingRect(forBounds bounds: CGRect) -> CGRect {

        return bounds.inset(by: padding)

    }

}

UIViewController+ATCAAdditions

import UIKit

extension UIViewController {

    func hideKeyboardWhenTappedAround() {

        let tap: UITapGestureRecognizer = UITapGestureRecognizer(target: self,
        action:

        #selector(UIViewController.dismissKeyboard))

        tap.cancelsTouchesInView = false

        view.addGestureRecognizer(tap)
    }
}

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้


```

    }

    @objc func dismissKeyboard() {

        view.endEditing(true)

    }

}

UIColor

import UIKit

class HelperDarkMode {

    static let mainThemeBackgroundColor: UIColor = {

        if #available(iOS 13.0, *) {

            return UIColor { (traitCollection: UITraitCollection) -> UIColor in

                switch traitCollection.userInterfaceStyle {

                    case

                        .unspecified,

                        .light:

                            return .white

```

```

        case .dark:

            return UIColor(hexString: "#252525")

        @unknown default:

            fatalError()

    }

}

} else {

    return UIColor.white

}

}

}

static let mainThemeForegroundColor: UIColor = UIColor(hexString: "#0997E8")

static let mainTextColor: UIColor = {

    if #available(iOS 13.0, *) {

        return UIColor { (traitCollection: UITraitCollection) -> UIColor in

            switch traitCollection.userInterfaceStyle {

                case

                    .unspecified,

```

```

        .light: return UIColor(hexString: "#464646")

        case .dark: return UIColor(hexString: "#b9b9b9")

        @unknown default:

            fatalError()

    }

}

} else {

    return UIColor(hexString: "#464646")

}

}

}

static let mainSubtextColor: UIColor = {

    if #available(iOS 13.0, *) {

        return UIColor { (traitCollection: UITraitCollection) -> UIColor in

            switch traitCollection.userInterfaceStyle {

                case

                    .unspecified,

                    .light: return UIColor(hexString: "#999999")

```

```

        case .dark: return UIColor(hexString: "#666666")

        @unknown default:

            fatalError()

    }

}

} else {

    return UIColor(hexString: "#999999")

}

}

static let statusBarStyle: UIStatusBarStyle = .default

static let hairlineColor: UIColor = {

    if #available(iOS 13.0, *) {

        return UIColor { (traitCollection: UITraitCollection) -> UIColor in

            switch traitCollection.userInterfaceStyle {

                case

                    .unspecified,

```

```

        .light: return UIColor(hexString: "#d6d6d6")

        case .dark: return UIColor(hexString: "#292929")

        @unknown default:

            fatalError()

    }

}

} else {

    return UIColor(hexString: "#d6d6d6")

}

}0

static let colorGray0: UIColor = {

    if #available(iOS 13.0, *) {

        return UIColor { (traitCollection: UITraitCollection) -> UIColor in

            switch traitCollection.userInterfaceStyle {

                case

                    .unspecified,

                    .light: return .black

```

```

        case .dark: return .white

        @unknown default:

        fatalError()

    }

}

} else {

    return .black

}

}

static var colorGray3: UIColor = {

    if #available(iOS 13.0, *) {

        return UIColor { (traitCollection: UITraitCollection) -> UIColor in

            switch traitCollection.userInterfaceStyle {

                case

                .unspecified,

                .light: return UIColor(hexString: "#333333")

                case .dark: return UIColor(hexString: "#cccccc")

```



```
@unknown default:

fatalError()

}

}

} else {

return UIColor(hexString: "#333333")

}

}

static var colorGray9: UIColor = {

if #available(iOS 13.0, *) {

return UIColor { (traitCollection: UITraitCollection) -> UIColor in

switch traitCollection.userInterfaceStyle {

case

.unspecified,

.light: return UIColor(hexString: "#f4f4f4")

case .dark: return UIColor(hexString: "#0b0b0b")

}

}

}

}

}
```

@unknown default:

เอกสารนี้เป็นเอกสารที่ส่งงานไว้สำหรับการเรียนเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และตัว 765 ่างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

        return UIColor(hexString: "#f4f4f4")

    }

}

} else {

    return UIColor(hexString: "#f4f4f4")

}

}0

static func darkModeEnabled() -> Bool {
    return {

        if #available(iOS 13.0, *) {

            let color = UIColor { (traitCollection: UITraitCollection) -> UIColor in

                switch traitCollection.userInterfaceStyle {

                    case

                        .unspecified,

                        .light: return .white

                    case .dark: return .black

                }

            }

            @unknown default:

```

```

        return .white

    }

}

return color.toHexString() == "#000000"

} else {

    return false

}

}0
}
}

extension UIColor {

    convenience init(hexString: String, alpha: CGFloat = 1.0) {

        let hexString: String = hexString.trimmingCharacters(in:
CharacterSet.whitespacesAndNewlines)

        let scanner = Scanner(string: hexString)

        if (hexString.hasPrefix("#")) {

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```
scanner.scanLocation = 1
```

```
}
```

```
var color: UInt32 = 0
```

```
scanner.scanHexInt32(&color)
```

```
let mask = 0x000000FF
```

```
let r = Int(color >> 16) & mask
```

```
let g = Int(color >>
```

```
& mask
```

```
let b = Int(color) & mask
```

```
let red = CGFloat(r) / 255.0
```

```
let green = CGFloat(g) / 255.0
```

```
let blue = CGFloat(b) / 255.0
```

```
self.init(red:red, green:green, blue:blue, alpha:alpha)
```

```
}
```

```
func toHexString() -> String {
```

```
    var r:CGFloat = 0
```

```
    var g:CGFloat = 0
```

```
    var b:CGFloat = 0
```

```
    var a:CGFloat = 0
```

```
    getRed(&r, green: &g, blue: &b, alpha: &a)
```

```
    let rgb:Int = (Int)(r*255)<<16 | (Int)(g*255)<<8 | (Int)(b*255)<<0
```

```
    return String(format:"#%06x", rgb)
```

```
}
```

```
}
```

```
UIButton
```

```
import UIKit
```

```

extension UIButton {

    func configure(color: UIColor = .blue, font: UIFont =
UIFont.boldSystemFont(ofSize: 12)) {

        self.setTitleColor(color, for: .normal)

        self.titleLabel?.font = font

        self.layer.borderColor = color.cgColor

        self.layer.borderWidth = 1.0

        self.layer.cornerRadius = 5.0
    }

    func configure(icon: UIImage, color: UIColor? = nil) {

        self.setImage(icon, for: .normal)

        if let color = color {

            tintColors = color

        }

    }

    func configure(color: UIColor = .blue,

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้


```

font: UIFont = UIFont.boldSystemFont(ofSize: 12),

cornerRadius: CGFloat,

borderColor: UIColor? = nil,

backgroundColor: UIColor,

borderWidth: CGFloat? = nil) {

self.setTitleColor(color, for: .normal)

self.titleLabel?.font = font

self.backgroundColor = backgroundColor

if let borderColor = borderColor {

self.layer.borderColor = borderColor.cgColor

}

if let borderWidth = borderWidth {

self.layer.borderWidth = borderWidth

}

self.layer.cornerRadius = cornerRadius

}

}

```

UIImage

```

import UIKit

extension UIImage {

    static func loadImage(_ name: String, template: Bool = false) -> UIImage {

        var image = UIImage(named: name)!

        if template {

            image = image.withRenderingMode(.alwaysTemplate)

        }

        return image

    }

}

UITextField

import UIKit

extension UITextField {

    func configure(color: UIColor = .blue,

        font: UIFont = UIFont.boldSystemFont(ofSize: 12),

        cornerRadius: CGFloat,

```

```

        borderColor: UIColor? = nil,

        backgroundColor: UIColor,

        borderWidth: CGFloat? = nil) {

    if let borderWidth = borderWidth {

        self.layer.borderWidth = borderWidth

    }

    if let borderColor = borderColor {

        self.layer.borderColor = borderColor.cgColor

    }

    self.layer.cornerRadius = cornerRadius

    self.font = font

    self.textColor = color

    self.backgroundColor = backgroundColor

}

}

```

ATCClassicSignUpViewController

```
import UIKit
```

```

class ATCClassicSignUpViewController: UIViewController {

    @IBOutlet var titleLabel: UILabel!

    @IBOutlet var containerView: UIView!

    @IBOutlet var backButton: UIButton!

    @IBOutlet var nameTextField: ATCTextField!

    @IBOutlet var phoneNumberTextField: ATCTextField!

    @IBOutlet var passwordTextField: ATCTextField!

    @IBOutlet var emailTextField: ATCTextField!

    @IBOutlet var errorLabel: UILabel!

    @IBOutlet var signUpButton: UIButton!

    private let tintColor = UIColor(hexString: "543931")

    private let backgroundColor: UIColor =
HelperDarkMode.mainThemeBackgroundColor

    private let textFieldColor = UIColor(hexString: "#B0B3C6")

    private let textFieldBorderColor = UIColor(hexString: "543931")

```

```
private let titleFont = UIFont.boldSystemFont(ofSize: 33)

private let textFieldFont = UIFont.systemFont(ofSize: 18)

private let buttonFont = UIFont.boldSystemFont(ofSize: 23)

override func viewDidLoad() {

    super.viewDidLoad()

    view.backgroundColor = backgroundColor

    let color = UIColor(hexString: "#282E4F")
    backButton.tintColor = color

    backButton.addTarget(self, action: #selector(didTapBackButton), for:
.touchUpInside)

    titleLabel.font = titleFont

    titleLabel.text = "Sign Up"

    titleLabel.textColor = tintColor

    nameTextField.configure(color: textFieldColor,

font: textFieldFont,
```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```
cornerRadius: 0,

borderColor: textFieldBorderColor,

backgroundColor: backgroundColor,

borderWidth: 1.0)

nameTextField.placeholder = "Full Name"

nameTextField.clipsToBounds = true

emailTextField.configure(color: textFieldColor,

font: textFieldFont,

cornerRadius: 0,

borderColor: textFieldBorderColor,

backgroundColor: backgroundColor,

borderWidth: 1.0)

emailTextField.placeholder = "E-mail Address"

emailTextField.clipsToBounds = true

phoneNumberTextField.configure(color: textFieldColor,

font: textFieldFont,
```


cornerRadius: 0,

borderColor: textFieldBorderColor,

backgroundColor: backgroundColor,

borderWidth: 1.0)

phoneNumberTextField.placeholder = "Phone Number"

phoneNumberTextField.clipsToBounds = true

passwordTextField.configure(color: textFieldColor,

font: textFieldFont,

cornerRadius: 0,

borderColor: textFieldBorderColor,

backgroundColor: backgroundColor,

borderWidth: 1.0)

passwordTextField.placeholder = "Password"

passwordTextField.isSecureTextEntry = true

passwordTextField.clipsToBounds = true

signUpButton.setTitle("Submit", for: .normal)

```
signUpButton.addTarget(self, action: #selector(didTapSignUpButton), for:
.touchUpInside)
```

```
signUpButton.configure(color: backgroundColor,
```

```
font: buttonFont, cornerRadius: 10,
```

```
backgroundColor: UIColor(hexString: "543931"))
```

```
self.hideKeyboardWhenTappedAround()
```

```
}
```

```
override func viewWillAppear(_ animated: Bool) {
```

```
super.viewWillAppear(animated)
```

```
self.navigationController?.setNavigationBarHidden(true, animated: false)
```

```
}
```

```
override func viewWillDisappear(_ animated: Bool) {
```

```
self.navigationController?.setNavigationBarHidden(false, animated: false)
```

```
}
```

```

@objc func didTapBackButton() {

    self.navigationController?.popViewController(animated: true)

}

@objc func didTapSignUpButton() {

    let signUpManager = FirebaseAuthManager()

    if let email = emailTextField.text, let password = passwordTextField.text {

        signUpManager.createUser(email: email, password: password) {[weak self]
(success) in

            guard let self = self else { return }

            var message: String = ""

            if (success) {

                message = "User was sucessfully created."

                let loginVC = ATCCClassicLoginScreenViewController(nibName:

"ATCCClassicLoginScreenViewController", bundle: nil)

```

```

        self.navigationController?.pushViewController(loginVC,    animated:
true)

    } else {

        message = "There was an error."

    }

    let alertController = UIAlertController(title: nil, message: message,
preferredStyle: .alert)

    alertController.addAction(UIAlertAction(title: "OK", style: .cancel,
handler: nil))

    self.display(alertController: alertController)
}
}

func display(alertController: UIAlertController) {

    self.present(alertController, animated: true, completion: nil)

}

}

```

ATCClassicLandingScreenViewController

```
import UIKit

class ATCClassicLandingScreenViewController: UIViewController {

    @IBOutlet var logoImageView: UIImageView!

    @IBOutlet var subtitleLabel: UILabel!

    @IBOutlet var titleLabel: UILabel!

    @IBOutlet var loginButton: UIButton!

    @IBOutlet var signUpButton: UIButton!

    private let backgroundColor: UIColor = .white

    private let tint_color = UIColor(hexString: "543931")

    // private let subtitleColor = UIColor(hexString: "#464646")

    private let signUpButtonColor = UIColor(hexString: "7D8E2E")

    private let signUpButtonBorderColor = UIColor(hexString: "#B0B3C6")

    private let titleFont = UIFont.boldSystemFont(ofSize: 40)

    // private let subtitleFont = UIFont.boldSystemFont(ofSize: 18)

    private let buttonFont = UIFont.boldSystemFont(ofSize: 23)
```

```
override func viewDidLoad() {  
  
    super.viewDidLoad()  
  
    view.backgroundColor = HelperDarkMode.mainThemeBackgroundColor  
  
    logoImageView.image = UIImage.localImage("logofarmuto")  
  
    //logoImageView.tintColor = tintColors  
  
    titleLabel.font = titleFont  
    titleLabel.text = "FARMUTO"  
    titleLabel.textColor = UIColor(hexString: "543931")  
  
    //    subtitleLabel.font = subtitleFont  
  
    //    subtitleLabel.text = "Start control your home."  
  
    //    subtitleLabel.textColor = HelperDarkMode.mainTextColor  
  
    loginButton.setTitle("Log in", for: .normal)
```



```
loginButton.addTarget(self, action: #selector(didTapLoginButton), for:
.touchUpInside)

loginButton.configure(color: .white,

font: buttonFont,

cornerRadius: 20,

backgroundColor: tintColor)

loginButton.addTarget(self, action: #selector(didTapLoginButton), for:
.touchUpInside)

signUpButton.setTitle("Create Account", for: .normal)

signUpButton.addTarget(self, action: #selector(didTapSignUpButton), for:
.touchUpInside)

signUpButton.configure(color: signUpButtonColor,

font: buttonFont,

cornerRadius: 20,

borderColor: signUpButtonBorderColor,

backgroundColor: backgroundColor,

borderWidth: 1)
```

```

        signUpButton.addTarget(self, action: #selector(didTapSignUpButton), for:
        .touchUpInside)

    }

    override func viewWillAppear(_ animated: Bool) {

        super.viewWillAppear(animated)

        self.navigationController?.setNavigationBarHidden(true, animated: false)

    }

    override func viewWillDisappear(_ animated: Bool) {

        self.navigationController?.setNavigationBarHidden(false, animated: false)

    }

    @objc private func didTapLoginButton() {

        let loginVC = ATCCClassicLoginScreenViewController(nibName:

        "ATCCClassicLoginScreenViewController", bundle: nil)

        self.navigationController?.pushViewController(loginVC, animated: true)

    }

```

```

@objc private func didTapSignUpButton() {

    let    signUpVC    =    ATCClassicSignUpViewController(nibName:
"ATCClassicSignUpViewController",
bundle: nil)

    self.navigationController?.pushViewController(signUpVC, animated: true)

}

}

ATCClassicLoginScreenViewController

import FBSDKCoreKit

import FBSDKLoginKit

import FirebaseAuth

import UIKit

class ATCClassicLoginScreenViewController: UIViewController {

    @IBOutlet var titleLabel: UILabel!

    @IBOutlet var passwordTextField: ATCTextField!

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

@IBOutlet var contactPointTextField: ATCTextField!

@IBOutlet var loginButton: UIButton!

@IBOutlet var separatorLabel: UILabel!

@IBOutlet var facebookButton: UIButton!

@IBOutlet var backButton: UIButton!

private let backgroundColor = HelperDarkMode.mainThemeBackgroundColor

private let tint_color = UIColor(hexString: "543931")

private let titleFont = UIFont.boldSystemFont(ofSize: 33)

private let buttonFont = UIFont.boldSystemFont(ofSize: 23)

private let textFieldFont = UIFont.systemFont(ofSize: 18)

private let textFieldColor = UIColor(hexString: "#B0B3C6")

private let textFieldBorderColor = UIColor(hexString: "543931")

private let separatorFont = UIFont.boldSystemFont(ofSize: 16)

private let separatorTextColor = UIColor(hexString: "#464646")

เอกสารนี้เป็นเอกสารสงวนลิขสิทธิ์สำหรับการใช้งานเพื่อการศึกษาเท่านั้น เมื่ออนุญาตให้นำไปใช้ประโยชน์ด้านการค้า

ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```
override func viewDidLoad() {  
  
    super.viewDidLoad()  
  
    view.backgroundColor = backgroundColor  
  
    backButton.setImage(UIImage.localImage("arrow-back-icon", template: true),  
for: .normal)  
  
    backButton.tintColor = UIColor(hexString: "#282E4F")  
  
    backButton.addTarget(self, action: #selector(didTapBackButton), for:  
.touchUpInside)  
  
    titleLabel.font = titleFont  
  
    titleLabel.text = "Log In"  
  
    titleLabel.textColor = tintColor  
  
  
    contactPointTextField.configure(color: textFieldColor,  
  
font: textFieldFont,  
  
cornerRadius: 0,  
  
borderColor: textFieldBorderColor,
```

backgroundColor: backgroundColor,

borderWidth: 1.0)

contactPointTextField.placeholder = "E-mail"

contactPointTextField.textContentType = .emailAddress

contactPointTextField.clipsToBounds = true

passwordTextField.configure(color: textFieldColor,

font: textFieldFont,

cornerRadius: 0,

borderColor: textFieldBorderColor,

backgroundColor: backgroundColor,

borderWidth: 1.0)

passwordTextField.placeholder = "Password"

passwordTextField.isSecureTextEntry = true

passwordTextField.textContentType = .emailAddress

passwordTextField.clipsToBounds = true

// separatorLabel.font = separatorFont


```
// separatorLabel.textColor = separatorTextColor

// separatorLabel.text = "OR"


loginButton.setTitle("Log In", for: .normal)

loginButton.addTarget(self, action: #selector(didTapLoginButton), for:
.touchUpInside)

loginButton.configure(color: backgroundColor,
font: buttonFont,
cornerRadius: 10,
backgroundColor: tintColor)

// facebookButton.setTitle("Facebook Login", for: .normal)

// facebookButton.addTarget(self, action: #selector(didTapFacebookButton), for:
.touchUpInside)

// facebookButton.configure(color: backgroundColor,

// font: buttonFont,

// borderRadius: 10,

// backgroundColor: UIColor(hexString: "#334D92"))
```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

self.hideKeyboardWhenTappedAround()

}

override func viewWillAppear(_ animated: Bool) {

    super.viewWillAppear(animated)

    self.navigationController?.setNavigationBarHidden(true, animated: false)

}

override func viewWillDisappear(_ animated: Bool) {

    self.navigationController?.setNavigationBarHidden(false, animated: false)

}

@objc func didTapBackButton() {

    self.navigationController?.popViewController(animated: true)

}

@objc func didTapLoginButton() {

    let loginManager = FirebaseAuthManager()

```

```
guard let email = contactPointTextField.text, let password =  
passwordTextField.text else { return  
}
```

```
loginManager.signIn(email: email, pass: password) {[weak self] (success) in
```

```
guard let self = self else { return }
```

```
var message: String = ""
```

```
if (success) {
```

```
let Storyboard = UIStoryboard(name: "Main", bundle: nil)
```

```
let vc = Storyboard.instantiateViewController(withIdentifier: "view")
```

```
vc.modalPresentationStyle = .fullScreen
```

```
self.present(vc , animated: true , completion: nil)
```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

    } else {

        message = "There was an error."

    }

    let alertController = UIAlertController(title: nil, message: message,
preferredStyle: .alert)

    alertController.addAction(UIAlertAction(title: "OK", style: .cancel, handler:
nil))

    self.display(alertController: alertController)

}

}

@objc func didTapFacebookButton() {

```

```

let loginManager = LoginManager()

loginManager.logIn(permissions: ["Email"], from: self) { (result, error) in

    if error != nil {

        self.showPopup(isSuccess: false)

        return

    }

    guard let token = AccessToken.current else {

        print("Failed to get access token")
        self.showPopup(isSuccess: false)
        return
    }

    FirebaseAuthManager().login(credential:
FacebookAuthProvider.credential(withAccessToken:
token.tokenString)) {[weak self] (success) in

        self?.showPopup(isSuccess: true)

    }

}

```

```

}

@objc func pushToHome() {

let Storyboard = UIStoryboard(name: "Main", bundle: nil)

let vc = Storyboard.instantiateViewController(withIdentifier: "view")

vc.modalPresentationStyle = .fullScreen

present(vc , animated: true , completion: nil)

}

func display(alertController: UIAlertController) {
self.present(alertController, animated: true, completion: nil)
}

}

extension ATCClassicLoginScreenViewController {

func showPopup(isSuccess: Bool) {

let successMessage = "User was sucessfully logged in."

```



```

let errorMessage = "Something went wrong. Please try again"

let alert = UIAlertController(title: isSuccess ? "Success": "Error", message:
isSuccess ?

successMessage: errorMessage, preferredStyle: UIAlertController.Style.alert)

alert.addAction(UIAlertAction(title: "Done", style: UIAlertAction.Style.default,
handler: nil))

self.present(alert, animated: true, completion: nil)

}
}
AppDelegate
import Firebase
import UIKit
import FacebookCore

@UIApplicationMain

class AppDelegate: UIResponder, UIApplicationDelegate {

var window: UIWindow?

```

```
func application(_ application: UIApplication, didFinishLaunchingWithOptions  
launchOptions:
```

```
[UIApplication.LaunchOptionsKey: Any]?) -> Bool {
```

```
    FirebaseApp.configure()
```



```
// Window setup
```

```
window = UIWindow(frame: UIScreen.main.bounds)
```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องแจ้งถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

```

window?.rootViewController = UINavigationController(rootViewController:
ATCCClassicLandingScreenViewController(nibName:
"ATCCClassicLandingScreenViewController",
bundle: nil))

window?.makeKeyAndVisible()

return true
}

func application(_ app: UIApplication, open url: URL, options:
[UIApplication.OpenURLOptionsKey : Any] = [:]) -> Bool {
return AppDelegate.shared.application(app, open: url, options: options)
}
}

```