中央研究院
資訊科學研究所
Institute of Information Science, Academia Sinica • Taipei, Taiwan, ROC

TR-IIS-05-007

User Scenarios and Designs of Smart Pantry, Object Locator and Walker's Buddy: Consumer Electronics for the Elderly

Jane W. S. Liu, B. Y. Wang, C. W. Hsueh, and Y. H. Liao
Institute of Information Science
Academia Sinica, Taiwan

C. S. Shih, T. W. Kuo, and A. C. Pang
W. H. Chen, Y. T. Liu, and H. C. Yeh,
Department of Computer Science and Information Engineering
National Taiwan University, Taiwan

T. S. Chou and C. Y. Huang
Department of Computer Science
National Tsing-Hua University, Taiwan

J. K. Zao
Department of Computer Science and Information Engineering
National Chiao-Tung University

July 2005 || Technical Report No. TR-IIS-05-007
http://www.iis.sinica.edu.tw/LIB/TechReport/tr2005/tr05.html

Institute of Information Science
Academia Sinica, Taiwan

Technical Report TR-IIS-05-007

User Scenarios and Designs

Of

Smart Pantry, Object Locator and Walker's Buddy:

Consumer Electronics for the Elderly

Jane W. S. Liu, B. Y. Wang, C. W. Hsueh, and Y. H. Liao
Institute of Information Science
Academia Sinica, Taiwan

C. S. Shih, T. W. Kuo, and A. C. Pang
W. H. Chen, Y. T. Liu, and H. C. Yeh,
Department of Computer Science and Information Engineering
National Taiwan University, Taiwan

T. S. Chou and C. Y. Huang
Department of Computer Science
National Tsing-Hua University, Taiwan

J. K. Zao
Department of Computer Science and Information Engineering
National Chiao-Tung University

Abstract

In the recent decade, declining birth rate and increasing average life span have led to significant growth in the over-60 fraction of population in developed and developing countries. One can easily deduce from this trend the growing need and business opportunity for consumer electronic appliances and services designed to improve the quality of life and safety of the elderly. Representative devices of this kind include smart pantry, object locater and walker's buddy. A *smart pantry* holds household supplies, inventories its own contents, and automates the purchasing and delivery of the supplies stored in it. An *object locater* can help its user find personal and household items such as keys, glasses and remote control. A *walker's buddy* scans pavement and alerts the user of steps and obstacles ahead. While such devices are motivated by the needs and desires of the elderly, busy and absent-minded people of all ages can also enjoy their services.

The report describes typical user scenarios of these devices. It also describes and evaluates alternative designs. The designs using only today's technologies are less than ideal in usability and robustness. The advances needed to significantly improve the devices are in near horizon. The report identifies the shortfalls in technology for each of the devices.

Chapter 1 Introduction

This progress report describes the usages and designs of three representative consumer electronic appliances for the elderly. We refer to this class of appliances and services collectively as SISARL, which stands for *Sensor Information Systems* (*Services*) *for Active Retirees and Assisted Living.* Declining birth rate and increasing average life span in developed and developing countries have led to significant growth in the elderly segment of global population. The average percentage population over 65 in many of these countries will soon exceed the average percentage population under 15 [1, 2]. From this trend, one can easily see the growing need and business opportunity for SISARL.

Recent technological advances have made a wide range of SISARL appliances and services feasible. Some of them are consumer electronic products designed primarily for enhancement of quality of life. Representative SISARL devices of this type include smart pantry, object locater and walker's buddy. A *smart pantry* holds household supplies, such as laundry detergent and shampoo. It monitors its own contents, automates the purchasing and delivery of the objects in it, and thus relieves its user from the chore of keeping essential supplies at hand. An *object locater* can help its user find personal and household items such as keys, glasses and remote control. Anyone who has ever misplaced such objects when they are needed can appreciate the utility of a locater. A *walker's buddy* scans pavement and alerts the user of steps and obstacles ahead. In this way, the device can help a walker or jogger avoid hazards in the path. While the devices are motivated by the needs and desires of the elderly, busy and absent-minded people of all ages can also enjoy their services. Other SISARL appliances can serve as assistive and point-of-care devices in assisted-living and home-care environments. An example is medicine dispensers. A *medicine dispenser* reminds its user at times medications should be taken, monitors and controls their dosages each time and notifies a caretaker when the user fails to take some medication in time. Medicine dispensers and other examples of SISARL devices can be found in [3, 4].

The report describes typical user scenarios of smart pantry, object locater, and walker's buddy. It also describes and evaluates alternative designs. The designs using only today's technologies are less than ideal in usability and robustness. The advances needed to significantly improve the devices are in the near horizon. The report identifies them.

Following this introduction, Chapter 2 discusses common assumptions underlying the devices described here. Chapters 3, 4, and 5 describe smart pantries, object locaters and walker's buddy, respectively. Chapter 6 summarizes related efforts and future work. The appendix contains descriptions of preliminary implementations of smart pantries.

Chapter 2 Common Assumptions

Some of the first generation SISARL appliances are already available today. As examples, one can get object locaters and medicine dispensers in specialty stores and over the web. Many other SISARL appliances can be built from off-the-shelf building blocks and platforms. Smart pantry is an example. Additional examples of first generation SISARL appliances can be found in [3].

As subsequent chapters will point out, the designs of existing object locaters and medicine dispensers ignore many important issues, including usability, customizability, modularity and extensibility. By dealing with these issues, we can be improved the devices significantly.

The first generation designs documented here make two kinds of common assumptions. One concerns with users, and the other concerns with the required technologies and infrastructures. An observation is that the time for first generation SISARL is now as an increasingly larger number of baby boomers begin to retire or will retire within this decade. Most of them are still in good health and live active life styles. Some still work. SISARL is not essential yet for them. They may use some appliances for reasons of convenience, safety and health maintenance; they may acquire and enjoy health and safety related products, much like one acquires and enjoys fancy phones and other consumer electronic products. The SISARL products discussed here are targeted for this segment of the elderly population. As an active retiree ages and relies on some appliances more or more heavily for assistance in daily activities, the appliances will become more and more critical and evolve into assisted living and home care devices. With improving health and increasing life expectancy of retirees, this transition may take ten to twenty years.

Whenever possible, the first generation SISARL appliances make use of existing end-user interface devices. There are many advantages. Take cell or cordless phones for example. A phone already provides keyboard, display, and multimedia capabilities. One can count on continued improvements in their processing and storage capabilities and audio and display qualities. Adding several SISARL functions to the devices is feasible and is a way to reduce the cost of SISARL. More importantly, the user will not have to carry multiple gadgets and learn to use them. From usability standpoint, the fewer distinct user interfaces, the better.

It is reasonable to suppose that the effective lifetime of an appliance ranges from five to twenty years. Bandwidth requirements of first-generation appliances are relatively small. Take vital sign monitors for example. The bit rate for telephone quality voice is sufficient for encoding of data collected by majority of vital sign sensors. Their sensor-data processing requirements can be readily met by current and future common platforms. None of the appliances described here

make use of video. Many competing wireless standards (e.g., Bluetooth and Zigbee) will work for interconnection within a household. There will be an increasing number of acceptable choices with different tradeoffs in the future. It is important for the current designs to be open for adoption of new choices.

In contrast, information technological infrastructures, social services, personnel care culture, and health care and delivery practices will not change significantly in the next five years. In particular, a significant percentage of households will not have computers and Internet access during this period of time. Only relatively affluent households will have broadband Internet connections. For this reason, the first generation appliances described here require only dial-up connections to the outside world. At the same time, they must be extensible so that users with computers, Internet and broadband linkage can benefit from these facilities now. Extensibility is also essential to ensure smooth, glitch-free transition to the next generation systems.

Chapter 3 Smart Pantries

As stated in Chapter 1, a smart pantry is used to hold non-perishable house supplies such as paper towel, shampoo and detergent. A smart panty knows its contents and can automate their replenishments. For example, each of the pantries in Figure 3-1 knows that a 6-roll bag of paper towel is on the top shelf. When the last roll is removed, the pantry knows which store and what brand of towels to order and requests the store to deliver a new 6-roll bag by a user-specified date.

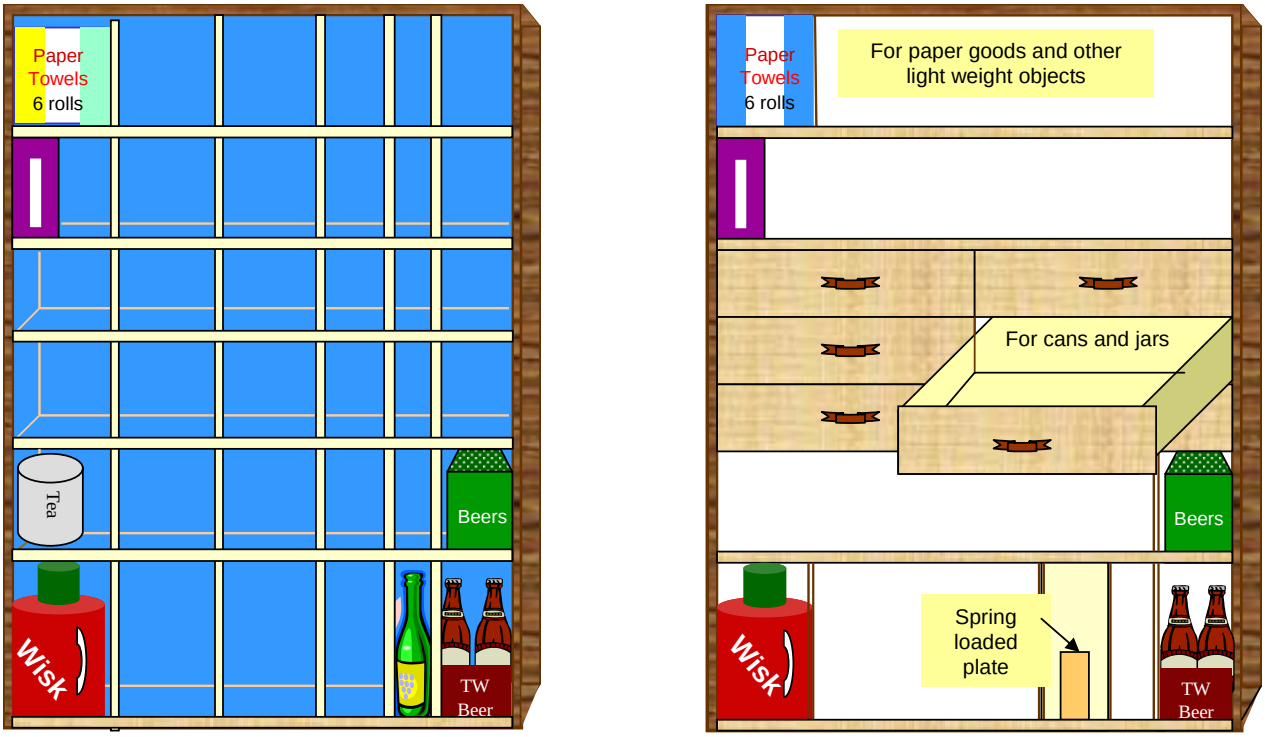

Figure 3-1 Smart Pantries

This chapter describes three versions of smart pantries and corresponding scenarios of their usage. Section 3.1 describes major differences, as well as commonalities, in all versions. Sections 3.2 -3.4 describe the versions. Section 3.5 is a summary.

3.1 Differences and Commonalities

The versions are the picture-id version, bar-code version and RFID version. The major difference amongst the versions arises from differences in the technologies used for object identification

and user interfaces required for improved usability and robustness. Figure 3-2 summarizes them.

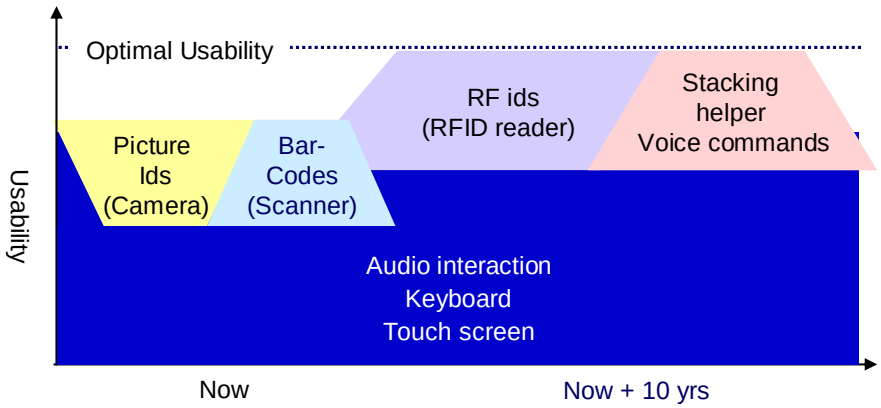

Figure 3-2 User Interface Alternatives

3.1.1 User Interface Alternatives

In the *picture-id version*, an overhead camera is used to capture the contents of the pantry. Objects in the pantry are identified by their pictures: Specifically, in each purchase order sent by the pantry to the supplier, each object is specified by its photo. The usability of this design is not ideal from the supplier's point of view. Reasons include that pictures in the order message may not clearly show the brands and sizes of objects and the supplier must first look up and enter the name or other id of each object in order to locate the object in the store. The version is easy to use from the pantry owner's point of view, however. The owner may have to deal with alert messages sent by the pantry on occasions such as when something blocks the camera's view and some pictures in the order received by the supplier are unclear. Otherwise, the pantry looks and feels to the owner like a dumb pantry.

In the *bar-code version*, each object is identified by its bar code. Because every object in every purchase order is uniquely identified by a bar code, this version is easy to use from the supplier's point of view. On the other hand, the user must scan the bar code of every object in the pantry. If the user neglects to scan some object when its supply is exhausted, the pantry will not be able to order it automatically. In this sense, the pantry is not robust.

In the future, when suppliers tag all objects with smart tags, a pantry equipped with a RFID reader can easily keep track of the objects and keep inventory on them as they are moved in and out of the pantry. This is the *RFID-version*. This version combines the advantages of the picture-id and bar-code versions: The owner can simply put objects in the pantry without concern

with letting the pantry know what they are, and the supplier is given unambiguous identifiers of objects to be delivered. Stacking helper in Figure 3-2 refers to some futuristic device that automates the placements of objects in the pantry. With this capability, the user will be relieved completely from the boring chores getting supplies and putting them away.

Figure 3-2 shows that all versions use keyboard, microphone, and voice recorder to support user-pantry interaction. (I/O devices may come with a phone or a computer, rather than dedicated to the pantry.) A smart pantry may also have a touch screen. Several user scenarios described in subsequent sections assume that a microphone and a voice recorder are configured to form an audio interface. The interface captures recorded voice of the user and plays back user voice interleaved with pre-recorded voice. In this way, the pantry carries out audio interaction with the user to make the pantry friendlier and more tolerant to misuse.

3.1.2 Other Assumptions

Compartments and Their Contents Like ordinary dumb pantries, the storage space in a smart pantry is divided into compartments. Figure 3-1 shows two alternative configurations. In the pantry shown in Figure 3-1(a), shelves are divided by vertical partitions. The picture-id version is constrained to use this pantry configuration. The fact that each compartment is clearly defined by a rectangular boundary of known dimensions simplifies the extraction of pictures of objects in the compartments from a picture of the entire pantry.

In contrast, shelves in the pantry in Figure 3-1(b) are not necessarily divided vertically. Rather, each compartment corresponds to a spring-loaded plate, similar to those often found on drug store shelves. A compartment is empty when the corresponding plate is at the front of the shelf, as exemplified by the plate on the bottom shelf in the figure. The compartment is nonempty when the plate is pushed to the back of the shelf by an object. This construction, plus binary sensors on the positions of the plates, enables the pantry to determine whether a compartment is empty. The bar-code version is constrained to use this configuration. The RFID version imposes no constrain on pantry configuration.

Throughout this chapter, we refer to compartments by 2-tuples of rows and columns. As examples, (1, 1) and (3, 4) refer to the compartment in the first (topmost) row and first (leftmost) column and the compartment in the third row and fourth column, respectively. We use *rows* and *columns* to mean number of rows and number of columns in the pantry.

Both picture-id and bar-code versions require that objects in each compartment are identical. The RFID version does not impose this constrain.

Suppliers and Network Access In subsequent descriptions, we assume that information (including user preferences) required for contacting suppliers, placing orders and arranging payments and deliveries were entered into the pantry at initialization time. The scenarios do not state explicitly how the pantry reaches the suppliers. The default is that orders are placed via fax or instant messages over a dial-up connection. A user with broadband internet access can configure the pantry to place orders via Internet.

3.2 Picture-Id Version

This section describes how the picture-id version of smart pantries may be used. The pantry handles placements, removals and ordering of objects in each compartment independently from that of the other compartments. In particular, it cannot tell whether objects in two or more compartments are identical. Consequently, it will order replenishment when the last of the objects in any compartment is removed and the compartment becomes empty, even though some other compartments may contain more of the same objects. The other versions of the pantry do not have this shortcoming.

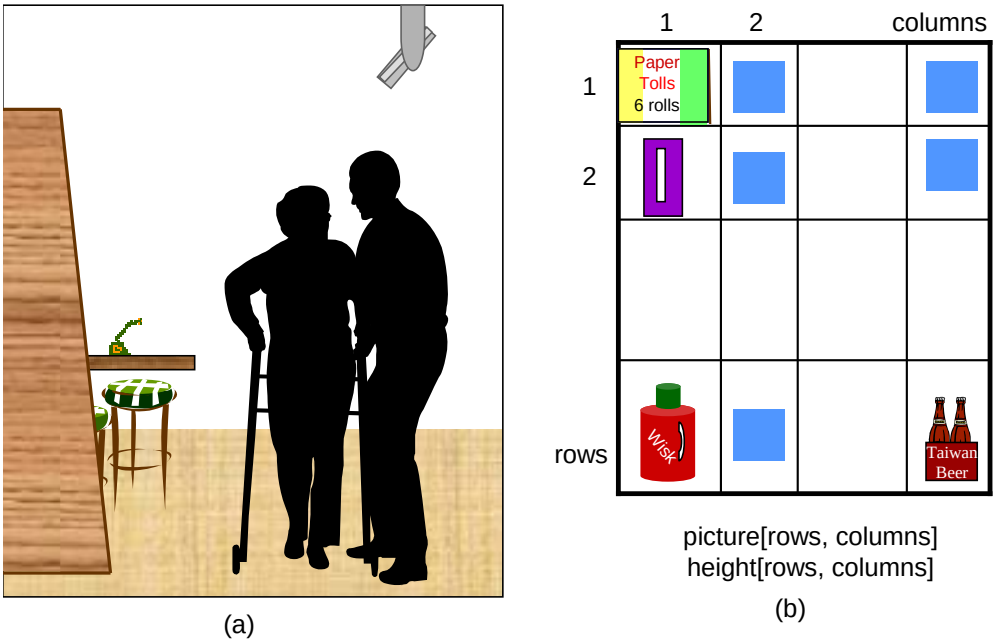

Figure 3-3 Physical Arrangement and Compartment Contents

Figure 3-3(a) shows a possible arrangement of the pantry and camera so that the camera can capture a front view of the pantry. (The picture may look like Figure 3-1(a).) After processing the picture, the pantry generates an array of pictures, one for objects in each compartment, as exemplified by the array picture[rows, columns] in Figure 3-3(b). The blue box in the array is a

picture of an empty compartment. The figure shows that the pantry also keeps an array of compartment heights. By superimposing compartment dimensions with object images, the pantry can make it easier for the supplier to tell the sizes of the objects. For sake of brevity, we ignore this feature in our discussion hereafter.

Figure 3-4 shows physical components of the pantry. The sensor is the camera. The base unit contains processing and storage components that do most of the work. The camera and the base unit are mounted overhead. The keyboard, microphone and touch screen must be within easy reach of the user. They are in the remote unit. The units are connected wirelessly.

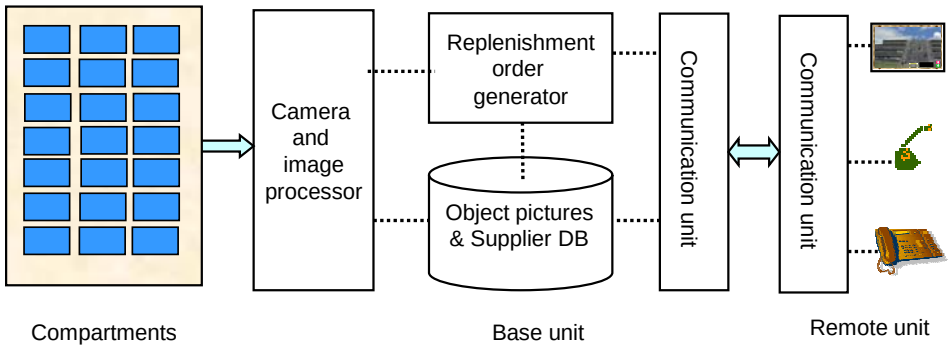

Figure 3-4 Architecture of Picture-Id Version

Figure 3-5 describes two scenarios of use. All scenarios in this chapter assume that there are multiple users. Without loss of generality, we assume that there are two users and call them Alice and Bob.

We can see that the users can freely place objects into compartments and remove all or part of contents of one or more compartments. The pantry captures the changes in compartment contents periodically. It extracts the picture of the object in each non-empty compartment (i, k) and stores the picture at picture[i, k] as a record. Whenever the compartment becomes empty, the pantry picks up the picture at picture[i, k], places the picture in a purchase order and sends the order to a designated supplier. A pseudo-code description of the thread that does this work can be found in Section A.1 of the appendix.

An examination of the second scenario at the bottom of Figure 3-5 shows that an error would occur whenever the user empties the content of a compartment and places the content in another compartment. In this case, pantry would mistakenly think that the supply of the object is exhausted and generate and issues an order for its replenishment. This kind of error can be prevented if the user is asked to confirm every order before the pantry sends it or is provided

with a means to tell the pantry that a swap of contents is taking place. An audio interface and touch panel can be used for this purpose. We will illustrate how these interfaces may be used in user scenarios for the bar-code version, which is described in the next section.

As a part of preferences, a user may provide a time-to-delivery interval, i.e., the length of time from the time when a purchase order is sent to the time when delivery is desired. This interval is called *replenishment time*. Its value can be anywhere from ASAP (as soon as possible) to a week, a month, etc. Based on user's setting of replenishment time, the pantry and supplier may batch orders and deliveries for multiple objects. Because a picture-id version pantry cannot distinguish objects from each other, all objects in the pantry have the same replenishment time and default supplier. The other versions do not have this limitation.

Alice: Puts a package of paper towels in previously empty compartment (1, 2).
Pantry: Takes a picture of pantry and analyzes the picture.
Detects and handles empty-> nonempty transition of (1, 2):
- Sees (1, 2) is no longer empty,
- Generates a picture of the object in it, and
- Stores the picture at picture[1, 2].

Alice: Puts another package of paper towels in (1, 2) and a pack of Taiwan beer in empty compartment (5, 3).
Bob: Takes the last bottle of shampoo from (4, 4).
Pantry: Takes a picture of pantry and analyzes the picture.
Seeing (1, 2) still non-empty, updates the content stored at picture[1, 2].
Detects and handles empty->nonempty transition of (5, 3).
(As a consequence, picture[5, 3] contains a picture of Taiwan beer.)
Detects and handles nonempty->empty transition of compartment (4, 4):
- Sees (4, 4) becomes empty,
- Picks up the picture of the object removed from (4, 4) from picture[4, 4].
- Generates and queues an order containing the picture of the object.
- Change picture[4, 4] to (a picture of) empty (compartment).

Sends the supplier an order containing a picture of the object to be delivered.

Sometime late:
Bob: Swaps the contents of two compartments (2, 3) and (4, 6).
Pantry: Takes picture after the swap. Sees no nonempty->empty transition.
For each nonempty compartment, updates picture of the object in it.
(Now the pictures at picture[2, 3] and picture[4, 6] are that of the new contents.)

Figure 3-5 Placements and Removals in Picture-Id Version

3.3 Bar-code Version

Again, as its name implies, objects in a bar-code version are identified by their bar codes. With user's help, the pantry acquires incrementally the bar codes of all objects ever stored in it. Some of the date structures maintained by the pantry on objects are shown in Figure 3-6. The information on each object includes the bar code and a recorded voice description of the object. The compartments field gives the list of compartments holding the kind of object: The list is empty when the pantry contains no object of the kind. The user may choose to keep some kind of objects in multiple compartments and requests the pantry to order replenishment only when all the compartments holding them become empty. In that case, the field points to the list of all those compartments. We recall that with a picture-id version, the user is constrained to order all objects from the same supplier with the same replenishment time. A bar-code version allows these choices to be made individually. For this purpose, the pantry maintains supplier and user preference information on each kind of object.

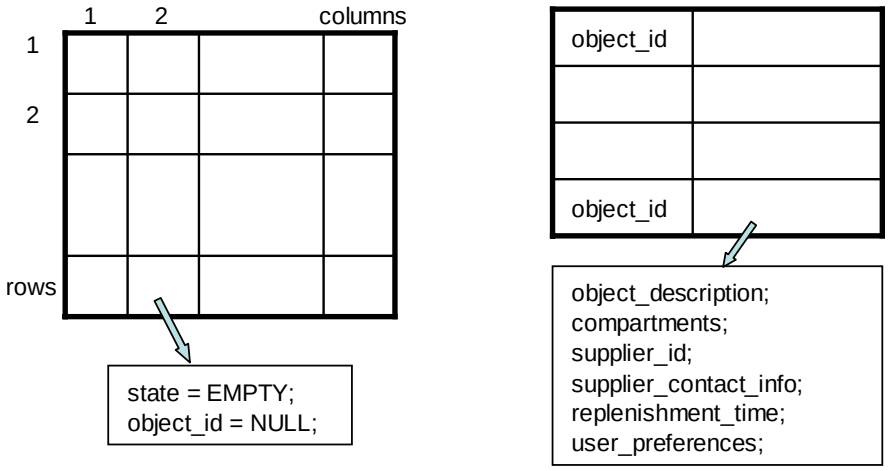

Figure 3-6 Compartment and Object Descriptions

Figure 3-7 shows the physical components of a bar-code version. For now, we assume that sensors used to monitor compartment contents are wired to the pantry processor. We will return later to describe a way to configure an existing dumb pantry with wireless sensors.

Like picture-id version, a user of a bar-code version can also place objects and remove them in any order. The pantry can order their replacements correctly provided that every object is scanned by the time the last of its kind is removed. The bar-code version can also work in *load-pantry* mode: When using this mode, the user scans each object and then puts it in pantry; this mode is less error-prone when there are multiple objects to be put away.

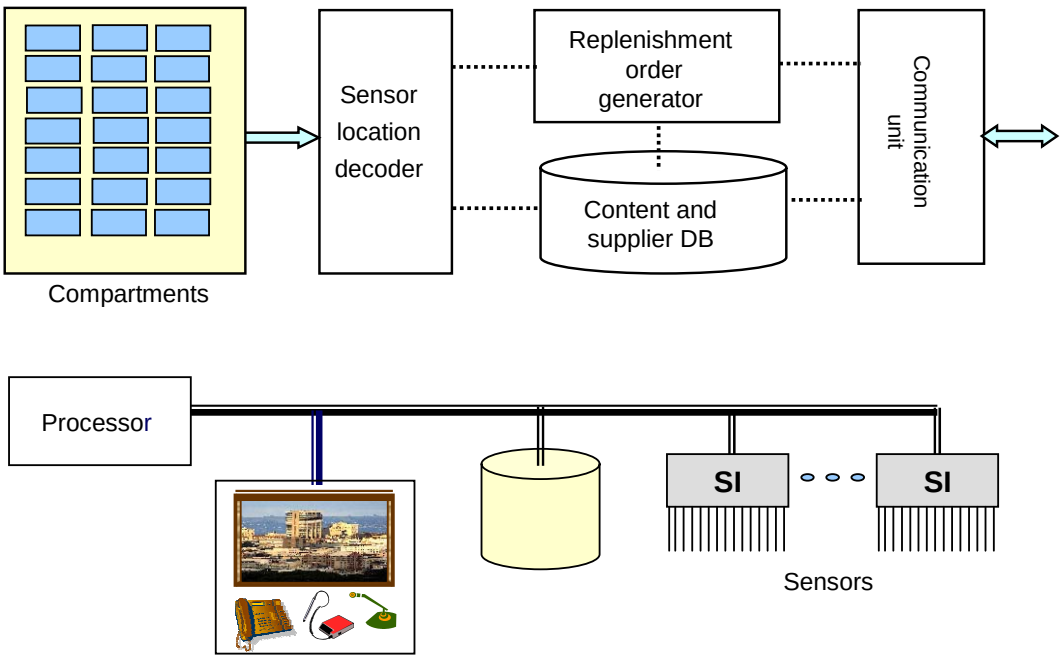

Figure 3-7 Architecture and Physical Components of Bar-Code Version

Figure 3-8 describes two scenarios: They illustrate user-pantry interactions during placement of objects into the pantry. In these and other scenarios for the bar-code version, lines describing user and pantry actions are in italic. Lines in regular font give dialogs between the user and the pantry. The object names are uttered in recorded user voice. They are interleaved with pre-recorded pantry voice in the dialogs. In the first scenario, the user and pantry cooperate in load-pantry mode: The user scans each object to provide the pantry with the bar-code of the object, gives a voice description of the object if the object is new to the pantry and then puts the object in a compartment when prompted by the pantry. The second scenario describes what may happen when the user puts an object in the pantry without providing the pantry with a bar code.

The scenario in Figure 3-9 illustrates removals of objects whose bar codes are known. As stated in Section 3.1, we assume that a supplier and replenishment time have selected for all objects. By default, the pantry will send to the supplier a purchase order and request the object be delivered within the replenishment time, unless the user cancels the order. A user can ignore the pantry in such interaction when the defaults are acceptable. The pantry may also be configured to assist the user to change the preferred supplier and replenishment time in the course of normal usage as illustrated here.

Figure 3-10 illustrates concurrent placements and removals of objects. The pantry continues to pay attention to placements of objects, ignoring removals of objects when their bar codes are

known. The pantry will reorder the objects from the supplier listed in its file. In contrast, when the bar code of a removed object is unknown, the pantry attends to the removal immediately, prompting the user to scan the removed object.

Putting away objects in load-pantry mode:

Bob: *Touches "**Load Pantry**" on touch screen to start putting two objects in pantry.*
Pantry: *Turns on bar code scanner.*
Bob: *Scans bar code of a pack of Taiwan Beer.*
Pantry: *Failing to find the bar code in database, says*: You just scanned an object. Tell me what it is.
Bob: Taiwan beer.
Pantry: *Records "Taiwan beer".* You said Taiwan beer. Please put it in a compartment.
Bob: *Puts the pack of beer in compartment (6, 4).*
Pantry: You have just put Taiwan beer in (6, 4), the middle compartment in bottom row.
Bob: *Scans a bag of paper towel*.
Pantry: *Finding the bar code and user's voice descriptions of the object, says:* You have a 6-roll bag of paper towel. Please put it away.
Bob: *Puts the paper towel in compartment (1, 2) and walks away.*
Pantry: You just put a 6-roll bag of paper towel in (1, 2).
Pantry: Shuts down bar code scanner and itself when timeout expires.

Some days later

Bob: *Touches "**Load Pantry**". Scans a replenished pack of Taiwan beer.*
Pantry: You have Taiwan beer. (6, 4) is still empty. You can put it there.
Bob: *Puts the beer in (6, 2).*
Pantry: You prefer (6, 2) for the Taiwan beer. *Records user preference.*
Bob and Pantry: *Continue to scan and put away objects collaboratively.*

Putting away object without bar code:

Alice: *Puts a 6-roll bag of paper towel in compartment (1, 3) and walks away.*
Pantry: *Sensing an object is put in (1, 3), turns on the bar code scanner*. You just put an object in compartment (1, 3). Please scan its bar code.
Bob: *Ignoring the pantry, puts a pack of Taiwan beer in (6, 4).*
Pantry: *Marks (1, 3) nonempty; leaves object_id for the compartment Null*, You just put an object in (6, 4). Please scan its bar code.
Bob: *Scans the pack of Taiwan beer*.
Pantry: (6, 4) now has Taiwan beer. *(In verbose mode)* Please scan the object in (1, 3)?
Bob: *Ignoring the pantry, walks way.*
Pantry: *Shuts down bar code scanner and itself when timeout expires*.

Figure 3-8 Placements of Objects in Bar-Code Version

Alice: *Remove the last bottle of beer in (6, 2).*
Pantry: You took the last of Taiwan beer from compartment (6, 2). Please touch "***Yes***" if you like it reordered, "***No***" if you do not want it reordered.
Alice: *Touches "**?**" to query about current object.*
Pantry: No other compartment has Taiwan beer.
Alice: *Touches "**Yes**" button.*
Pantry: *Noticing that user preference is not set for this object, displays suppliers **Costco**, **Jason's** and **Welcome***. Please select a supplier displayed on the screen.
Alice: *Touches "**Costco**".*
Pantry: I will order Taiwan beer from Costco and ask them to deliver it in (*default*) 3 days. If you want it sooner or later, please enter the number of days followed by the # sign now.
Alice: *Keys in "1#" on the keyboard of the nearby phone and walk away.*
Pantry: I will ask Costco to deliver Taiwan beer tomorrow. *Generates and sends order. Shuts down when timeout expires.*

Some days later after replenished supply of beer is delivered:

Alice: *Removes the last bottle of beer from (4, 3).*
Pantry: You took the last of Taiwan beer from (4, 3). I will reorder it from Costco and ask them to deliver it tomorrow. If this is not what you want, please touch "***No***".
Alice: *Walks away.*
Pantry*: Generates and sends order. Shuts down when timeout expires.*
Bob: *Removes the last bottle for beer from (6, 3).*
Pantry: You took the last of Taiwan beer from (6, 3). I will order Taiwan beer from Costco to be delivered tomorrow. If this not what you want, please touch "***No***".
Bob: *Touches "**No**".*
Pantry: Please touch "***Yes***" if you like it reordered.
Bob: *Touches "**Yes**".*
Pantry: *Displays suppliers **Costco, Jason's**, and **Welcome***. Please select a supplier displayed on the screen.
Bob: *Touches "**Welcome**".*
Pantry: I will order Taiwan beer from Welcome and ask them to deliver it in (default) 3 days. If you want it sooner or later, please enter the number of days followed by the # sign now.
Bob: *Walks away.*
Pantry: *Update user preferences. Sets time to generate and send order. Shuts down when timeout expires*.

Figure 3-9 Removals of Objects with Bar Codes

Giving higher priority to placement over removal of object with known id.

Alice: *In the middle of putting away supplies, scans a bag of paper towel.*
Pantry: You just scan an object, please tell me what it is.
Bob: *Removes the last bottle of beer from (4, 3) and walks away.*
Pantry: *Noticing the removal of an object with known id, continues to pay attention to the new object!*
Alice: Paper towels
Pantry: You said paper towels. Please put paper towels in a compartment.
Alice: *Puts the paper towels in (4, 3), the compartment left empty by Bob.*
Pantry: You just put paper towels in compartment (4, 3).
Pantry: *Switch attention to the removal.* You took the last of Taiwan Beer from (4, 3). I will reorder it from Costco and ask them to deliver it tomorrow. If this is not what you want, please touch "***No***" now.
Alice: *Walks away from the pantry.*
Pantry: *Generates and sends order for Taiwan beer. Shuts down when timeout expires*

Giving higher priority to removal of object with unknown id.

Alice: *Another day, in the midst of putting away supplies in load-pantry mode, scans a 6-roll bag of paper towel for the first time.*
Pantry: You just scan an object, please tell me what it is.
Bob: *Removes the last of object with unknown id from (4, 3).*
Pantry: *Noticing the id of the removed object is unknown, leaves load-pantry mode and attends to the removal.* You just removed the last of the object in (4, 3). I do not know what it is. Please scan it now.
Bob: *Lets Alice scan the removed object (Taiwan beer) and walks away.*
Pantry: You took the last of Taiwan beer from (4, 3). I will reorder it from Costco and ask them deliver it tomorrow. If this is not what you want, please touch "***No***" now; otherwise, touch "***Resume*** " to return to load-pantry mode.
Alice: *Hurrying now, puts the paper towel in (4, 3) and walks away*.
Pantry: *When wait_for_resume timeout expires, assumes that the object in (4, 3) is unrelated to the one Alice scanned earlier.* You just put an object in (4, 3). Please scan it.
Pantry: *When timeout expires, marks (4, 3) nonempty, object_id unknown. Generates and sends order for beer. Shuts down after a while.*

Figure 3-10 Concurrent Placements and Removals of Objects in Bar-Code Version

Errors during placements and removals are inevitable. Some errors are recoverable, as illustrated by the scenarios in Figure 3-11. The first one points out that the user may not follow the normal sequence of scan and placement. A consequence is that some objects in the pantry have no bar code identifiers. This error must be handled by the time when the objects are removed and replenishments are to be ordered. Figure 3-11 illustrates a way: The pantry asks the user to scan the object at the time of the removal if the pantry does not have the object's bar code.

The second scenario illustrates that how the pantry may have a wrong bar code. In this case, only the user can recognize the error and initiate corrective actions.

Alice: *While in load-pantry mode, scans a 6-roll bag of paper towels.*
Pantry: You have a 6-roll bag of power towel. Please put it in a compartment.
Alice: *Scans a six-pack of Taiwan beer.*
Pantry: *Overwrites the bar code of paper towel.* You have Taiwan beer, please put it in a compartment.
Alice: *Puts the paper towels in compartment (4, 3).*
Pantry: You put Taiwan beer in compartment (4, 3).
Alice: *Touches "**Back**" to continue in load-pantry mode. Takes power towels from (4, 3). Puts Taiwan beer in (4, 3).*
Pantry: You have Taiwan beer in compartment (4, 3).
Alice: *Puts power towel in compartment (1, 2) and walks away.*
Pantry: *Exits load-pantry mode.* You just put an object in compartment (1, 2), please scan its bar code.
Pantry: *When wait_for_bar_code timeout expires, marks (1, 2) non-empty and object_id unknown. Shuts down after a while.*

Some other time

Alice: *While in load-pantry mode, scans a 6-roll bag of paper towel.*
Pantry: You have power towel. Please put it in a compartment.
Alice: *Scans a pack of Taiwan beer.*
Pantry: *Overwrites bar code of paper towels.* You have Taiwan beer, please put it in a compartment
Alice: *Puts paper towels in compartment (4, 3) & Taiwan beer in (1, 2). Walks away.*
Pantry: You put Taiwan beer in compartment (4, 3). *Marks (4, 3) nonempty & object-id as bar code of Taiwan beer.*
Panty: You just put an object in compartment (1, 2), please scan its bar code.
Pantry: *When timeout expires, marks (1, 2) non-empty & object_id unknown. Wait a little longer and then shuts down.*

Later on

Bob: *Removes Taiwan beer from (1, 2).*
Pantry: *Mistaken the object in (1, 2) to be unknown, prompts Bob to scan the object.*
Bob and Pantry: *Proceeds to process removal of unknown object as illustrated earlier.*

Alice: *Removes paper towels from (4, 3).*
Pantry: You have removed the last of Taiwan beer in (4, 3). I will …..
Alice: *Realizing erroneous object_id, touches "**Back**" to initialize corrective action.*

Figure 3-11 Examples of Recoverable Errors and Error Recovery

We say that an error is unrecoverable when it causes the pantry to fail in ordering replenishment in time. Unfortunately, unrecoverable errors can occur. Figure 3-12 illustrates some of the ways for them to occur. In each of these situations, the pantry informs the user the

information it has on the content of the compartment. The user has a chance to correct the error if the voice of the pantry is heard.

Section A.2 in the appendix describes in c-like pseudo code two ways to implement a bar-code version of smart pantry. The more complex one uses three separate threads: One processes object placements, one processes removals of objects with bar codes, and the third one processes removals of objects without bar code. This implementation should yield better responsiveness, at least in principle. It also offers an excellent example for experimentation with approaches and tools for verification and testing. The simplest implementation uses a single thread to process placements and removals.

Alice: *Removes paper towels from (4, 3) while the recorded object-id is Taiwan beer. Walks away immediately.*
Pantry: You have removed the last of Taiwan beer in (4, 3). I will … and so on.
Pantry: *When timeout expires, generates and sends order for Taiwan beer, and unknowingly, leaves Alice and Bob short of paper towel.*

On another occasion

Bob: *Removes the last of object with unknown id from (2, 4).*
Pantry: You just removed the last of the object in compartment (2, 4). I do not know what it is. Please scan it now.
Bob: *Walks away.*
Pantry: *When timeout expires, records the time of removal of unknown object from (2, 4) and shuts down.*

Alice or Bob: *Later on, come to scan an object or remove an object.*
Pantry: *Flashes "**Apology**".* I am unable to order the object you removed from (2, 4) at 12:45 PM yesterday because I do not know what it is. *Proceeds to handle the error or current scan or removal according to the user's wish.*

Figure 3-12 Examples of Unrecoverable Errors of Bar-Code Version

To conclude this section, we note that the sensor in the picture-id version monitors the pantry remotely. (So is the monitor in the RFID version.) Consequently, there is no need to construct a smart panty from scratch. One can simply add a camera or a RFID reader and pantry electronics to a dump pantry to get a smart pantry. In contrast, sensors in the bar-code version described are on the shelves. They are wired to sensor interfaces as shown in Figure 3-7. This construction is reasonable for pantries specially built to be smart. Wiring up an existing pantry with sensors would be an unattractive option. A better alternative is to monitor compartment states wirelessly. A way is to make use of RFID's as described below.

Figure 3-13 describes the parts needed for the purpose. The happy face on the top of the pantry represents a RFID reader. It is mounted in the front of the pantry. The front and top views of spring-loaded plates are shown in the upper right part of the figure. Each plate consists of three parts: a plastic plate (shown in orange), a metal piece (shown in grey pattern) at the bottom of the plate, and a passive RFID tag (shown as a red star) at the center of the metal piece. Each shelf in the pantry has a front strip. The lower right part of the figure shows its construction. At the front is a strip (shown in orange), which is made of the pantry shelf material. Aligned with each spring-loaded plate, there is a metal piece on the back of the strip. The metal piece has a hole, which is sufficiently large to enclose a RFID tag.

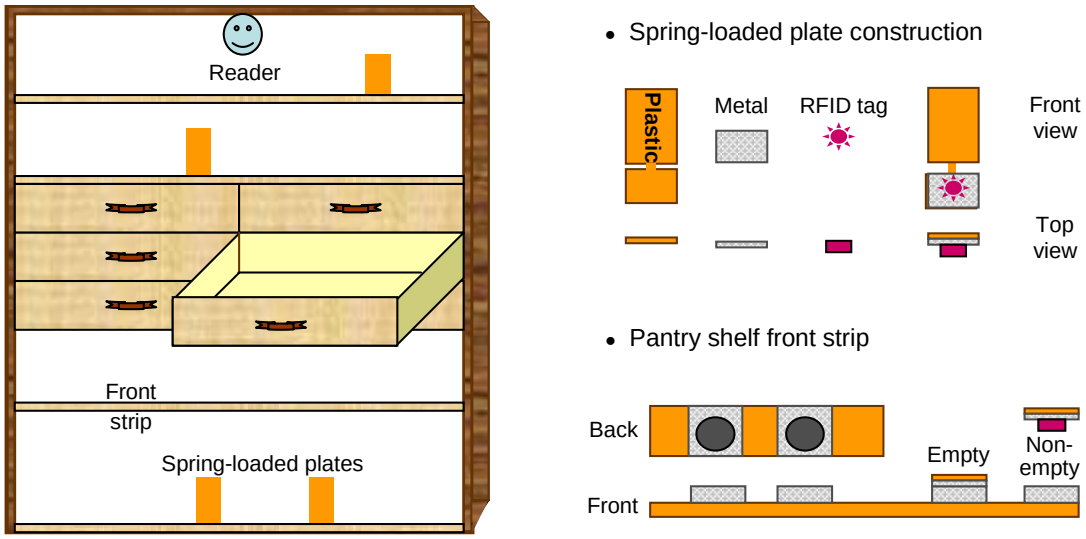

Figure 3-13 Components for Wireless Sensing of Compartment States

The lower right corner of Figure 3-13 shows that when a compartment is empty, the corresponding plate is pushed to the front of the shelf. The RFID tag at the bottom of the plate is enclosed in the metal piece on the plate and the metal piece on the front strip of the shelf. The tag becomes invisible to the reader. On the other hand, when the compartment is nonempty, the corresponding plate is pushed towards the back of the shelf. As a consequence, the RFID tag on the plate is exposed to the reader. Therefore, the pantry can determine the states of the compartments by having the RFID reader poll the RFID tags. The pantry controller knows that a compartment is nonempty if the reader can read the tag on the corresponding spring loaded plate; otherwise, the compartment is empty.

Figure 3-14 shows that the parts illustrated by Figure 3-13 can come separately from the pantry cabinet. The figure shows a smart pantry shelf containing three spring loaded plates and hence three compartments. A user can get individual shelves such as the one shown here, place

them on the shelves of an existing dumb pantry and, thus, turn the dumb pantry into a smart one.

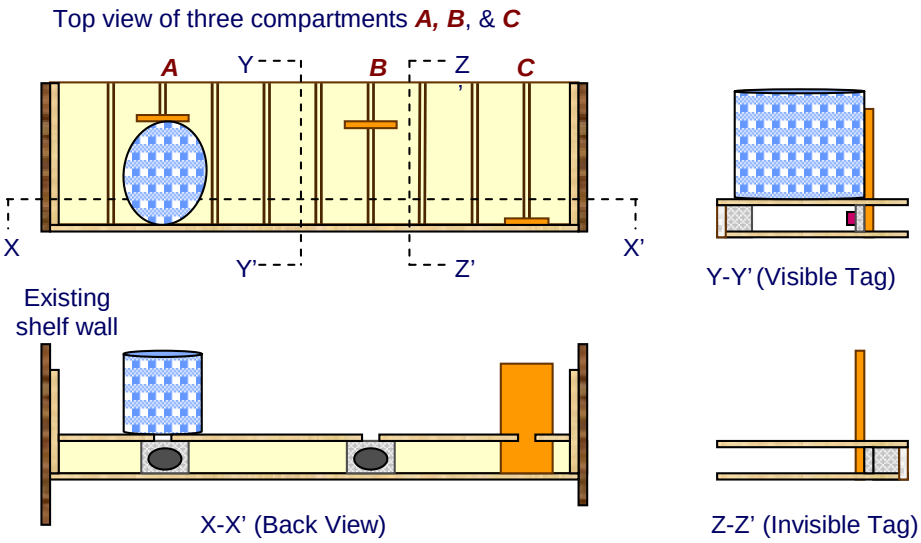

Figure 3-14 Separate Shelves for Making a Dumb Pantry Smart

3.4 RFID Version

By monitoring their ids, a RFID version can keep track of objects placed in the pantry, detects when the supplies of some objects are exhausted and automatically orders them. The user is relieved from the chore of making sure all objects are identified. The supplier has in every purchase order a unique identifier for every object, making it easy to locate the object within the store in preparation for its delivery. In addition to these advantages, the user of a RFID version can put different types of object in the same compartment. Moreover, it is straightforward to extend the design so that the pantry can keep track the remaining quantities of objects. (For example, rather than whether there is still a 6-roll bag of paper towel, the pantry knows exactly how many rolls there are left in the pantry.)

Figure 3-15 shows a possible configuration of such a pantry. We note that except for the difference in the sensors used, this version closely resembles the picture-id version. In this version, the sensor is a RFID reader. It either periodically polls the tags of all objects in the pantry or reads the tag of each object as it is being moved in and out of the pantry. A potential problem with periodic polling is interferences among tags: their responses may not be intelligible when too many tags respond at the same time. This problem is well known, and there are already some solutions. Another problem is visibility of the tags. Metal objects such as cans and aluminum foils may shield some RFID tags from the reader. This problem can be overcome by proper placement of the antenna and/or use of multiple antennae.

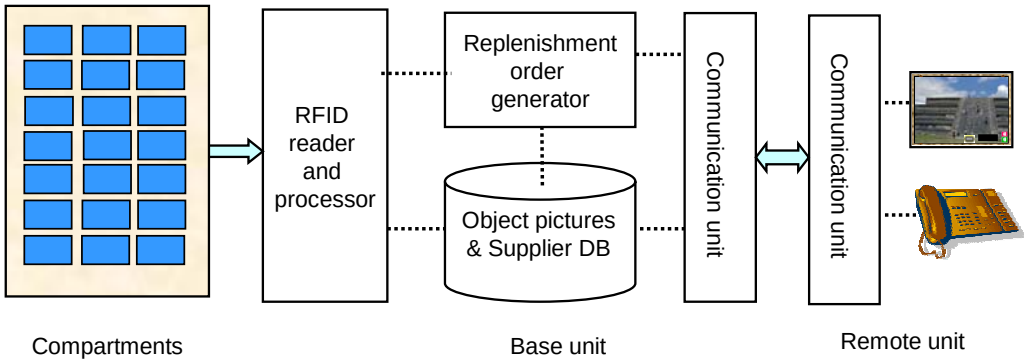

Figure 3-15 Architecture of RFID Version

The alternative of trying to read the id of every object when it is moved in and out of the pantry does not have these problems. If this is to be done without user assistance, some other means (e.g., an infra red beam) must be provided to detect movement of object across the front of the pantry and trigger the reader to capture the id of the object in transit. This presents a challenge and introduces a source of error since the time an object stays in the field of sight of the reader can be very short.

3.5 Summary

This chapter describes three versions of smart pantry. Picture id version is simple: One can get a smart pantry by adding a camera and pantry electronics (i.e., the base and remote units shown in Figure 3-4) to any dumb pantry that has compartments. The task of processing pantry picture to get object pictures is straightforward. The pantry owner can use it much like a dumb pantry when placing and removing objects. From the supplier's point of view, the version is far from ideal. Pictures included in the purchase orders sent by the pantry may not clearly identify the brand and size of each object, making it necessary to query the pantry owner for clarification. Moreover, the pictures must be translated to inventory control codes of the objects. This process is time consuming and error-prone if done manually, and the version does not have the image recognition capability required to automate the process. (Image recognition for this purpose is unlikely to be economical in the near future.)

In the RFID version, each object is unambiguously specified by an electronically readable id. The user can use it like a dumb pantry: Objects can be freely placed anywhere; different types of object can share a compartment. This version requires that all objects are individually tagged. This requirement is a roadblock [5], at least for now. While increasingly more suppliers will soon tag cases of goods, the cost of tags will remain too high for them to tag individual items.

From both the technical and usability points of view, the bar-code version represents a reasonable compromise. The supplier can rely on the bar codes in purchase orders to uniquely identify and easily locate the objects to be delivered. For the pantry to work well and error free, however, the users must follow the routine of scanning the content of each compartment at least once between the time when content is put in the pantry and when it is removed. For users who are willing to follow this routine, a bar-code version is sufficiently user friendly and reliable.

Chapter 4 Object Locaters

An electronic object location device, such as the ones offered by specialty stores (e.g., Sharper Image), contains a few (e.g., 8) transponders together with a key-box size interrogator. Each transponder can be attached to an object, such as a key chain or a reading glass. The *interrogator* has a button for each transponder. By pressing a button and causing the corresponding transponder to beep and flash, a user can locate the attached object within 40-60 feet from the interrogator. In subsequent discussion, we call such a device an (*object*) *locater* or (*object*) *location system* and call the components that are attached to objects *tags.*

Almost everyone has the experience of misplacing items big and small, from time to time, and can surely appreciate how useful such a simple device can be. For an individual who has become increasingly more forgetful, an object locater may become increasingly more essential. Both usability and cost of existing object locaters need significant improvement, however. $60 to 70 US dollars for a locater with capability for finding eight objects is too expensive. From usability point of view, a serious disadvantage is too few tags for too few items, especially since the device cannot be extended to handle more. An interrogator should be able to handle a larger number (e.g., 64 to 256) of tags. A user should be able to query for an object like one looks up and dials an entry in the address book of a phone. A bulky interrogator with one key per tag is not acceptable.

This chapter describes three alternative designs that improve over existing locaters in these respects. The first two designs use *active* (i.e., battery assisted) transponders as tags, one tag per object tracked by the locater. The third design uses passive RFID tags. (A *passive* tag has no battery. It derives the power it needs by rectifying the incident RF signal.) After describing the designs, the chapter discusses gaps in technology needed to realize the more ideal designs.

4.1 Commonalities

Like existing object location devices, the ones described here are primarily for locating inert small objects indoors. The interrogator is mobile: All three designs make use of a smart phone or PDA as an interrogator. In general, a location system may support multiple and heterogeneous interrogators. In a household with multiple cordless phones, any of the phones can serve as an interrogator. Similarly, the user can also use a computer for this purpose. For the sake of simplicity, the scenarios described here assume that only one interrogator is in use at a time. This restriction can be removed without difficulty for some designs.

The location system may come installed in a smart phone (or a PDA or computer) ready for initialization and use. Alternatively, the user may get the object location hardware, along with software that can be easily installed on some phones or a computer. One envisions that a relative small number (e.g., 8-16) of tags comes with the object locater system. The user can purchase additional tags, up to some maximum number (e.g., 256), as need for them arises.

All designs support query object, add object and delete object operations, as illustrated by Figure 4-1. In this example, the user attaches a tag to a key chain and then selects add object operation. In response, the locater registers the tag and prompts the user to enter a name (e.g., Key) to be associated with the tag (and the object attaching to the tag). Subsequently, the user can query for the location of the tag by selecting the query object operation followed by the name Key as illustrated by the two interrogators in the right-hand side of the picture. The user can remove the registration of a tag by selecting the detect object operation, followed by the name associated with the tag. The tag can now be used for some other object.

Figure 4-1 Add, Query and Delete Object Operations

The designs require the interrogators and tags to have communication range around 40-100 feet, or 15-30 feet, or 3-6 feet. We refer to these distances as *house-level*, *room-level*, and *desk-level* ranges, respectively.

4.2 HIT Locater

Again, tags in existing object location systems are transponders: They beep and flash in response. Because queries from the interrogator and responses from tags cover almost the entire house, we call a locater of this kind a *HIT* (*House-Level Interrogator and Tag*) System. Figure 4-2

illustrates how a HIT works. In this example, the user queries for a lost object in one of the bedrooms. The tag on the object beeps and flashes upon receiving the query signal. By following the sound from the tag, the user walks to the bedroom. Once there, flashing light from the tag also helps the user to locate the object.

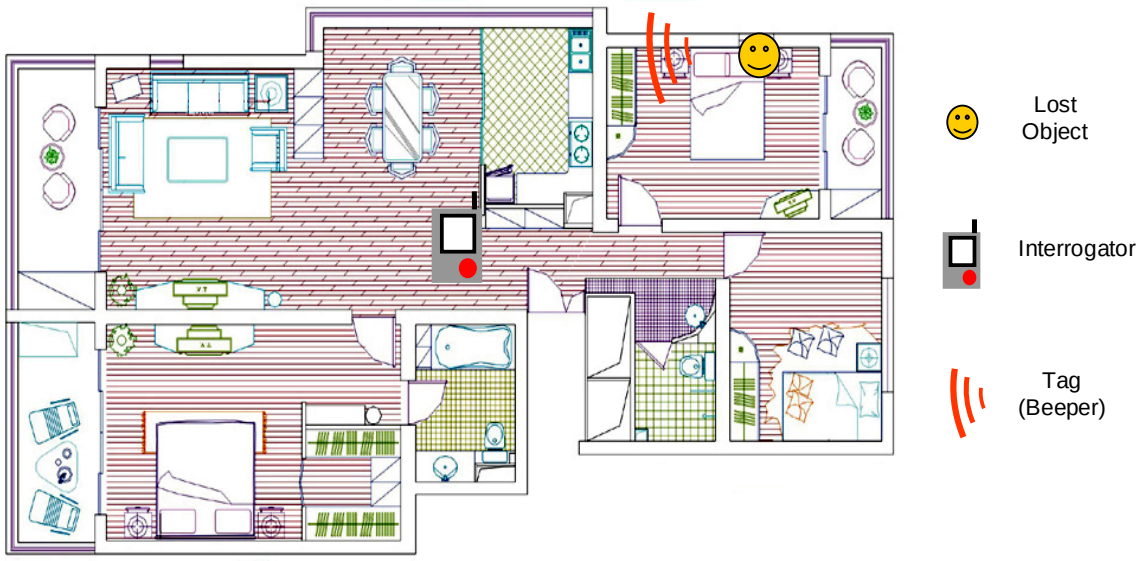

Figure 4-2 HIT Object Locater

One can get a HIT device with a sufficiently large capacity by extending the existing design in straightforward ways. For example, the 8-object location device offered by Sharper Image can be extended to handle 2^n tags by giving each tag an n-bit id encoded by a sequence of n or more RF pulses. The interrogator queries each object by broadcasting a preamble followed by id of the attach tag.

Every tag must process every query to determine whether the id in the query is its own. It is unlikely that the DC power obtained by rectifying the incident RF signal is sufficient for this purpose and to power the beeper and flash. Hence, the energy consumption of each tag increases linearly with the number of tags. Obviously, this is a serious disadvantage. In Figure 4-2, the user is in the entrance foyer, within hearing range from the lost object/tag. In a large house, the user may not be able to hear the tag. The lost object may be under some cover; so the user cannot see the tag flashes either. This is another serious disadvantage. Finally, there is no straightforward way to make a HIT locater work well when there are multiple interrogators. The designs described below are improvements over HIT.

4.3 RIAT Locater

Like a HIT, a *RIAT* (*Room-Level Interrogator, Agent and Tag*) object locater also uses active tags:

A RIAT tag is a RFID tag plus a beeper and flash light. In addition, RIAT employs a number of agents. An *agent* is a RFID reader/transponder. The read range of each agent covers a room-size area, which we call the *read region* of the agent.

As the scenario in Figure 4-3 illustrates, agents are placed strategically at fixed locations. In the figure, the read region of each agent is indicated by the blue circle around the agent. The union of the read regions of all agents includes the coverage area of the locater. During initialization, the locations of all the agents are entered into the interrogator. For this design, agent locations need not be precise. The general vicinity of each agent suffices. In this scenario, the location of each agent is specified by the name of the room where the agent is.

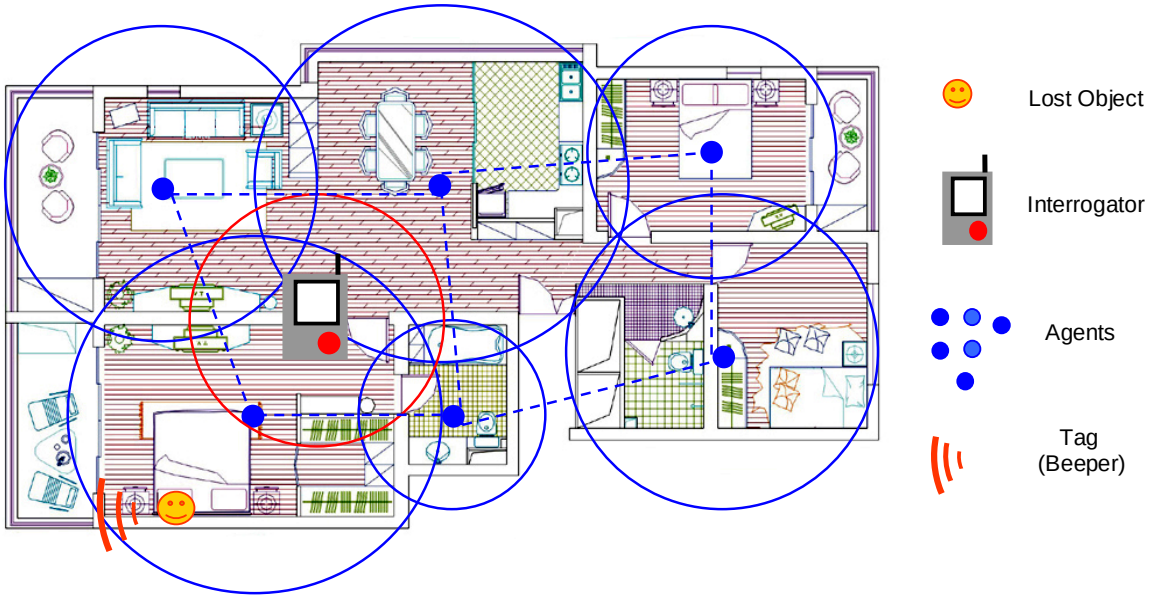

Figure 4-3 RIAT Object Locater

In response to a query containing an RFID, each agent reads the tags in its own region. When an agent finds the tag with the specified id, it responds positively to the interrogator. After receiving a positive response from an agent, the interrogator directs the user to the region covered by the agent. Figure 4-4 illustrates a way for the interrogator to do so. Upon reaching the area, the user can command the queried tag to beep and flash if the object is not in plain sight. If the interrogator receives no positive response before timeout (or receives negative responses), it reports to the user that queried object is not in the coverage area.

In the scenario, the interrogator also has room-level communication range. (In Figure 4-3, the transmission coverage area of the interrogator is indicated by the red circle around it.) It suffices for the interrogator to have a range large enough so that every query from anywhere within the coverage area can reach at least one agent. By keeping the communication range of the

interrogator small, its energy consumption is kept small.

Figure 4-4 Querying an object in RIAT

If agents are battery powered and have room-level transmission range, each query is routed from agent to agent. Similarly, responses from agents are routed collaboratively by agents back to the interrogator. When agents are connected to power source, there is no need to be concerned with their energy consumption. In this case, we may choose to give agents sufficient transmission power to reach the entire locater coverage area. The locater is likely to be more responsive when queries and responses are broadcast by agents. We can use one of the many existing broadcast schemes to keep the traffic at a minimum.

Tags in a RIAT are required to beep and flash sometimes, and this requires power. The energy consumption of RIAT tags is likely to be smaller than HIT tags, however, for the following reason: In RIAT, the agents partition the coverage area of the locater into smaller regions. While every tag in a HIT system must process every query, a tag is a RIAT is read in response to only a subset of queries.

Active RFID tags can be used as alternative to passive tags with beeper and flash. Unfortunately, the cost of active RFID tags is high, and they offer no significant advantage. Indeed, all choices of active tags are far from ideal because of the need for battery replacements, however infrequently.

RIAT requires RFID readers with room-level range. This is a demanding requirement [5]. Although the next generation RFID reader will have greater than 15 feet range, they are likely to work only under controlled conditions. More seriously, they are observed to degrade after each read, making them less reliable than required for object location.

4.4 DIAT Locater

DIAT (*desk-level, agent and tag*) locaters use only passive RFID tags. Agents in a DIAT are RFID readers/transponders. Agents are at fixed and known locations. Each agent can read tags in a desk-size region. The system employs a sufficient number of agents so that the union of their read regions fully covers the area where lost objects may be located. Figure 4-5 illustrates this configuration. Again, each blue circle indicates the read region of the agent at the center of the circle.

We note that the regions covered by agents overlap. A location within the coverage area of the locater can be identified by the subset of agents whose read regions include the location. To illustrate, we name the agents in the upper row A_1 through A_8. In Figure 4-5, the lost object is in the region where both A_1 and A_2 can read. In this scenario, the communication range of the interrogator is also desk-level, sufficient to reach at least one agent. The area reachable by transmissions from the interrogator is indicated by the red circle in the figure.

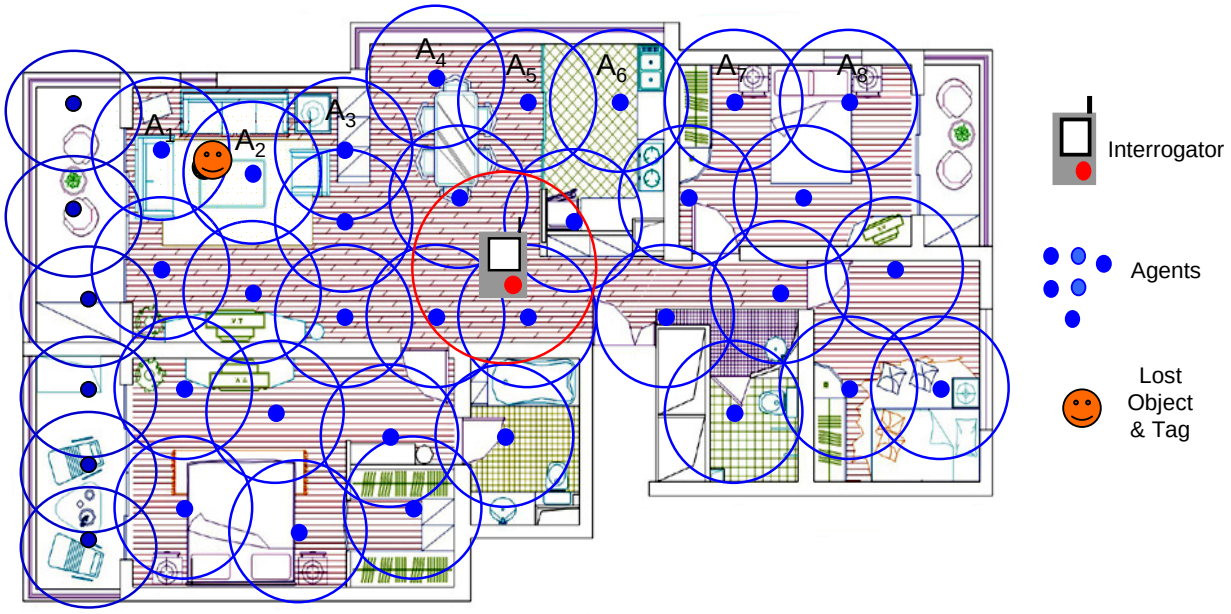

Figure 4-5 DIAT Object Locater

As in a RIAT system, each DIAT query and responses to the query are either routed or broadcast by agents until one or more agents responded or timeout expires. Based on the responses, the interrogator determines the location of the queried object. Figure 4-6 illustrate how the interrogator may display the information on object location: The user is told that the object is in the living room at location highlighted in yellow. Clearly, it is straightforward for the

interrogator to determine the vicinity of where the lost object is when it knows the locations of the agents who have found the tag on the object.

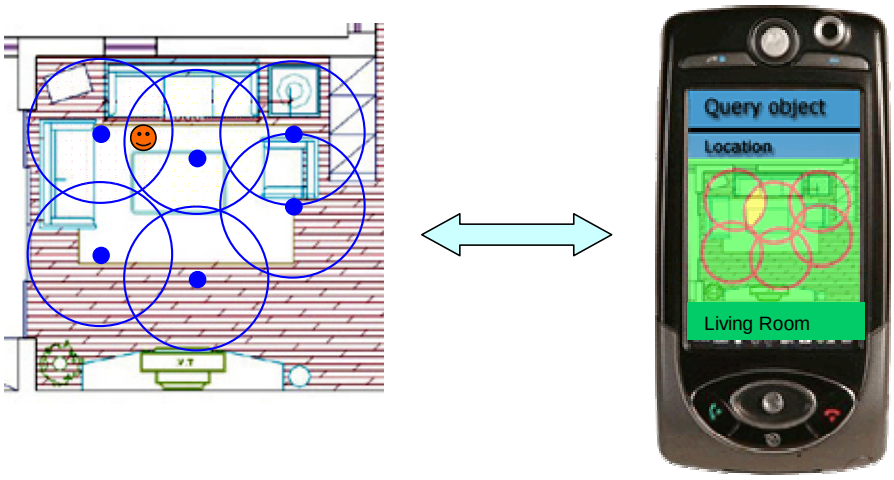

Figure 4-6 Location Display of a DIAT locater

The acquisition of agent location information is also straightforward if the interrogator can determine its own location accurately anywhere in the coverage area. To illustrate, let us suppose that the house is equipped with an indoor positioning system using which a mobile device like an interrogator can determine its position with accuracy in order of 0.5-2 inches. The interrogator provides register agent location and check agent location operations, which we will explain shortly. The user can install the agents and enter their locations into the system following the steps listed below:

1. Following the guidelines provided by the user manual, the user chooses a location in a furthermost corner of the house and proceed to install an agent at that location.
2. To install the agent, the user secures the agent physically. Ideally, the agent is plugged to a power source. The user then puts the interrogator next to the agent and selects register agent location operation. In response, the interrogator determines and stores its own location, generates an id for the agent, and associates the location with the agent id. (The agent id is used to support interrogator operations only.) It then prompts the user to optionally enter a name for the agent. The agent name need not be unique. For example, the user may name all agents in the living room Living Room.
3. Following the guidelines in the user manual, the user chooses for another agent a location within the read range of an agent that has previously been installed. The user then puts the interrogator at the chosen location and selects the check agent location operation. After determining its own location, the interrogator checks whether the read region of an agent at that location will near-optimally overlap with read regions of existing agents. If the new

location is a poor choice, the interrogator informs the user where a better location may be; otherwise, it instructs the user to put the next agent at the location. The user repeats this step if necessary until a good location is found.

4. After a new location is found, the user installs an agent at the location following the procedure outlined in step 2.
5. The user repeats steps 3 and 4 until all the agents are installed.

We note that close collaboration between the user and the locater system is essential when choosing locations for the agents. Without help, the user may put agents unnecessarily close together and, as a result, must purchase more than the necessary number of agents to provide full coverage. The agents should also not be placed too far apart and thus create blind regions. (Tags in a *blind region* cannot be read by any agent.) – In Figure 4-5, the right-hand side of the house has two blind regions. If the lost object were there, no agent would respond positively to a query for the object, and the interrogator would report that it cannot find the object. The effectiveness of the locater would be significantly compromised if blind regions are numerous and are scattered all over the house. (We can observe, however, that a small number of blind regions can be allowed as long as the user knows where the regions are. In our example, a negative response may nevertheless be informative to some extent: It tells the user to look for the object in the few blind regions in the house.)

The procedure outlined above is greedy: The location choices are made for one agent at a time on the basis of locations of agents already installed. No doubt, better location choices (in terms of a smaller number of agents and fewer blind regions) can be made if the interrogator were to compute near-optimal locations of all agents prior to installation based on global information of the coverage area. There is no shortage of algorithms for this computation, and they can be used as an alternative when accurate information on the coverage area (e.g. a blue print detailing the house floor plan and construction) is available.

4.5 Summary

This chapter describes the HIT, RIAT and DIAT designs of object locaters. HIT locaters are already available in stores, and the extensions described here can be built without technical difficulty. Tags in HIT process every query, and queried tag beeps and flashes when it detects its own id in a query. As the efficiency of RF rectifiers improves, tags will be able to derive from the query signal sufficient energy to do this work. Until then, they must be battery assisted. This is a major shortcoming of HIT.

RIAT offers a slight improvement over HIT design in regards with tag energy consumption. Nevertheless, tags still need batteries. An added expense comes from several 15-30-foot range RFID readers, each of which comes with a transponder, as agents. Such readers will not be available soon, at least not at a price appropriate for the object location application. An advantage RIAT has over HIT is that it is straightforward to extend the RIAT design to support multiple interrogators, while the increased complexity in query processing and responses by tags practically rules out this feature for HIT locaters.

DIAT offers the best choice among the designs. Like RIAT, it can be extended to support multiple interrogators. It uses only passive RFID tags and short-range RFID readers. These components bound to become cheaper as demand for them grows. Low-cost, easy to use ways to accurately determine interrogator location is a necessity. The procedure for installing DIAT agents described earlier assumes the existence of an indoor positioning system. For this purpose, indoor positioning systems such as Bat [6] and Cricket [7, 8] are possibilities. In particular, Cricket allows a mobile device equipped to receive RF and ultrasonic beacon signals to determine its position with centimeter accuracy. An alternative is to use floor with embedded sensors (e.g., in the Gator Tech Smart House [9]) for acquiring sufficiently accurate position. In both cases, unless the indoor position system is a shared infrastructure used by many smart appliances and services, its cost can be a forbidding factor.

Chapter 5 Walker's Buddy

Almost every one suffers occasional lapses in attention, misses a step, trips, and sometimes, even falls. When such a mishap happens to an elderly person, the consequence can be devastating. The walker's buddy described in this chapter is a portable device designed to help a walker (or jogger) minimize the chance of trip and fall by calling attention to the walker of common path hazards. After describing the intended usage of such a device, the chapter discusses alternative approaches to building the device.

5.1 User Scenario

Figure 5-1 illustrates how a walker's buddy may be used. In this picture, the buddy is a part of a smart phone or some other hand held device. The user carries it at the waist (e.g., in a phone case.) As the user walks, the device irradiates and illuminates the path ahead. It warns the user when it detects steps, bumps, or other hazards in the path.

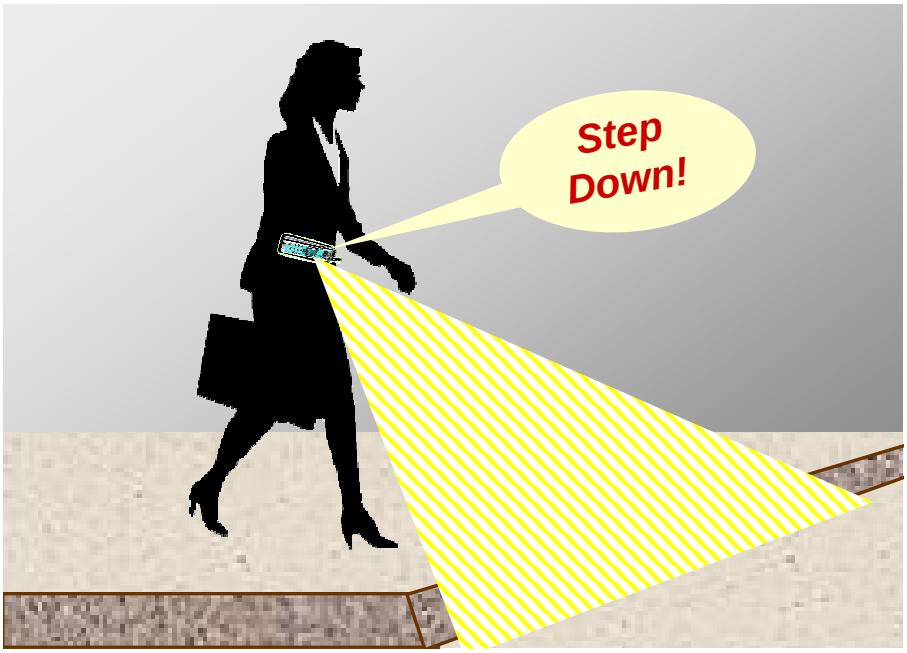

Figure 5-1 Example Illustrating Usage of Walker's Buddy

The picture shows a busy (and possibly absent minded) office worker walking to work or returning home. One also envisions an active elderly individual carrying it during morning or evening walks on city sidewalks or parks. For the device to be useful, it must work during day light hours, dust and night. It also must work anywhere. This requirement implies that the device

must be self contained. In particular, it should not rely on any pedestrian and vehicular service (such as electronic bulletins and messages from some intelligent transportation and pedestrian safety system) along the path to provide it with information on where hazards are.

The example assumes that the device is capable of distinguishing steps up or down and tells the user the downward step ahead. In general, a device may not be able to distinguish the types of hazard and issue specific directives. It can only provide general warnings, such as “watch your step”, which a companion is likely to say when the user walks with friends. Indeed, the user should be able to choose among many forms of warning sound, including recorded voices of friends who often walk, stroll or jog together.

A user may also want to customize the device in other aspects. A tunable parameter is *false alarm rate.* A cautious user may be willing to tolerate a higher false alarm rate and chooses to make the device highly sensitive (i.e., with a high hazard-detection rate). Other users may want to keep false alarm rate low at the expense of sensitivity. Warnings need to be issued sufficiently ahead of time before the user reaches a hazard. How far ahead is sufficient depends on the user. Therefore, *distance-to-hazard* (or *warning time*) is also a tunable parameter: Some user may want a warning delivered when the device detects a hazard five steps ahead. It is likely that the shorter the distance-to-hazard, the more accurate the device. So, a user who can react to warnings in two to three steps from hazards can choose a smaller distance-to-hazard for the advantage of a lower false alarm rate or higher detection rate. The device should be designed so that these and other tradeoffs can be customized to suite the user’s need and preferences.

5.2 Alternative Approaches

We note again that device must work under all lighting conditions, as well as anywhere, including places new to the user and the device. This requirement essentially eliminates vision as a viable technology. The remainder of the chapter discusses two alternative approaches to building this device: echo ranging and reflection pattern analysis.

Echo Ranging Asking many an engineer how to build a walker’s buddy, one is likely to hear “echo ranging”. In essence, the device is a range finder: It tries to find the distance from itself to the ground. Because steps, bumps and holes lengthen or shorten the distance, the device can distinguish flat surfaces from such hazards.

Echo ranging is a well developed technology. Both sonic and infra-red range finders are available. The required minimum and maximum distances of walker’s buddy are 2-4 feet and 10-20 feet, respectively; they can be met easily by many state-of-art range finders. A walker’s

buddy needs to detect changes in path surface of size an inch or smaller. This translates to a required accuracy of 0.5 – 1 % or better. This accuracy is somewhat demanding but nevertheless is not beyond reach of modern technology.

One can enumerate several potential problems of echo ranging when applied to walker's buddy. They include tolerance to environment noise, size and power consumption of the range-finder module, and so on. These problems are minor, however, when compared with range noise introduced by the movements of the user. Figure 5-2 explains. The walker's buddy sends inquiries periodically and measures the distance to the ground each period. Because the device moves up and down while the user walks or runs, the distance changes even when the surface of the path ahead is flat. This range variation is illustrated by the blue band and is called *movement noise* in the figure. It is likely that changes in range due to steps and hazards in the path are smaller than the peak movement noise. In principle, it is possible to predict user movements (hence, movement noise) and compensate for it. However, the added complexity can be significant, and it is difficult to achieve good compensation.

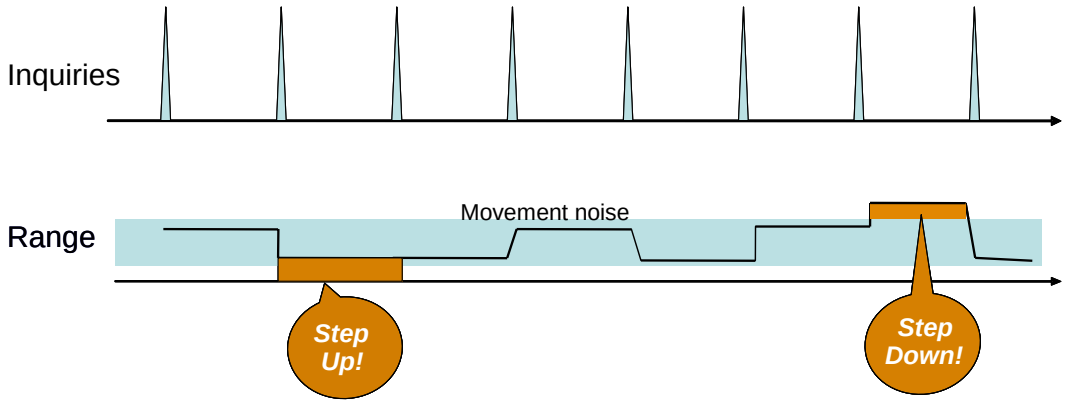

Figure 5-2 Range Noise Due to User Movement

Reflection Pattern Analysis The discussion above points out the fact that the information obtained by a range finder exceeds what the device needs for its purpose. A walker's buddy does not need to know accuracy distance to the ground. It suffices for the device to know that there is a change in the path surface ahead. This observation leads us to consider analysis of path reflection patterns. Rather than trying to get an accurate estimate of the distance from itself to the ground, the device examines the pattern of reflection and issues warnings when the pattern changed sufficiently.

Figure 5-3 illustrates this point. The inquiry beam from the walker's buddy should cover an area at least equal to the size of one step, and the sampling rate should be higher than once per

step. Consequently, there is no gap in the path surface examined by the device. Suppose that the delay spread of the reflected signal is wide enough for the device to measure and generate a rough histogram. We may see histograms of reflection delay similar to the ones depicted in the figure. The picture on the left is a histogram of reflection delay from a flat surface. It is more or less symmetrical, close to histograms from an even distribution. The other pictures show histograms of reflection delay from an upward step and a downward step. These histograms may not be symmetrical: The histogram of reflection delay from an upward step is likely to be narrower than that of a flat surface, and the fraction of short delays is larger. In contrast, the histogram of reflection delay from a downward step is likely to be wider, and the fraction of long delays is larger. The device compares delay histograms obtained during consecutive inquiries. When it detects non-negligible differences in the histograms, it assumes that the differences arise from some changes in path surface and issues a warning as a result.

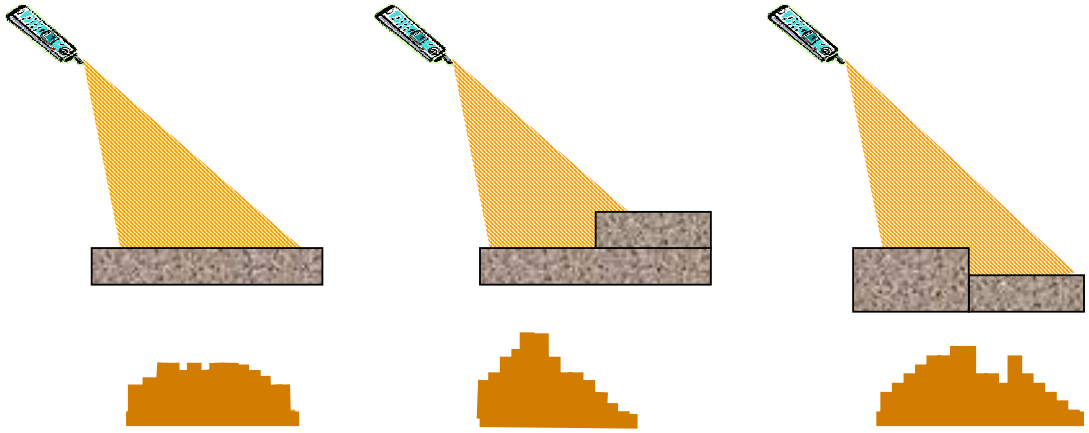

Figure 5-3 Multi-Path Reflection Patterns

Alternative to working with delay spread, a walker's buddy may base it decision on the intensity distribution of the reflection signal. Again, the device does not attempt to estimate how and by how much the path surface is about to change. It merely tries to detect non-negligible changes in minimum, maximum and average intensity or spectral density and warn the user when it notices any non-negligible change.

5.3 Summary

This chapter describes two techniques for implementing walker's buddy. The hand-held device is designed to warn its user of uneven surfaces ahead in the user's foot path that may trip the user. The viability of both techniques remains to be determined.

Echo ranging is likely to be a poorer technological choice because of the difficulty in estimating and compensating for user movement noise. The key assumption of reflection pattern analysis is that the device can measure the delay spread of reflected signals accurately enough to obtained distinct patterns. This assumption is valid if the inquiry is sonic. Since the distance from the device to the ground is around 3-6 feet, the spread in round trip delay is in order of 10 milliseconds. There is certainly time to examine the delay for patterns of interest in time of this order.

Chapter 6 Related and Future Works

The user scenarios and designs of the consumer electronic devices described here are parts of preliminary results of the SISARL project. Again, SISARL stands for Sensor Information Systems (Services) for Active Retirees and Assisted Living. We also use this term broadly to mean consumer electronic products, as well as assisted devices, for the elderly.

Similar to the numerous other efforts on assistive living technology (e.g., [9-17]), SISARL project is also motivated by increasing need and demand for products and services that can improve the quality of life of the elderly, increase their safety and security and enhance their dexterity and accessibility. The project complements the other efforts in two ways, however. First, most other efforts target as users individuals who are weak and sickly, physically and mentally. With tremendous improvement in health maintenance, the majorities of today's elderly are active and independent and likely remain so for all but one to three years of their lives. SISARL is primarily concerned with devices and services suitable for them. Consequently, we must consider usage scenarios and deal with design objectives and constraints typically ignored by other efforts. An example is privacy. A person truly in need of assistance by others may not mind being monitored intrusively, but active and independent elderly individuals may not tolerate any invasion of privacy. This is one of the reasons vision-based accident detection devices for homes are not in demand. Another example is support infrastructure. Most of elderly individuals live in homes of their younger years, and many of their homes are not equipped with computers and Internet, even in developed countries. This is why the devices described earlier require only dial-up connections. Smart floors and environments in future homes cannot be prerequisites for SISARL.

Second, the major thrust of our project is directed at developing the technology for the design and mass-production of SISARL devices and services. In contrast, most other efforts are primarily concerned with the development of new assistive technology. They have produced many promising prototypes with advanced capabilities such as motion and activity tracking, memory assistance, and personal health management. These results, together with advances in related areas, have made a wide range of SISARL capabilities feasible, but the results are not sufficient to bring high-quality devices and services with desired features and capabilities to the elderly. There must be methods and tools with which industry can design and build diverse SISARL products at affordable price, verify and test them to ensure their quality and dependability, and localize the products to compliant with different standards and regulations of different countries. Some SISARL devices may be in use for ten to twenty years. Their users

should be able to customize the devices to suite their changing needs and preferences, as well as evolving environment and support infrastructures. Easy to maintain and upgrade are essential for SISARL; otherwise, the devices will not “just work”. These issues are at the center of our attention.

As stated in previous chapters, advances in several areas will allow us to build better (i.e., more user-friendly, robust and tolerant of misuse) smart pantry, object locater and walker’s buddy. Specifically, the best (and possibility the most economical) smart pantry design awaits ultra-low-cost RFID technology [5], which will enable suppliers to individually tag every items. Ideally, object locaters use only passive tags, and they display on hand-held interrogators accurate locations of queried objects. This design requires some means (e.g., [7, 8]) to determine indoor locations sufficiently accurately (say within an inch). Indoor positioning systems are likely to be widely available soon.

The scenarios described in this report are the start of a library of user scenarios we are developing for SISARL applications. Currently, we are also creating and analyzing user scenarios of several other representative SISARL devices and services, including medicine dispenser, housekeeping helpers, and safety monitoring system. The results of this type of analysis allow us to derive realistic user requirements. The requirements in turn yield realistic design objectives and constraints of future products and services and criteria for evaluating the methods and tools used to build them. This approach to focusing our efforts on the right problems is termed user-centered approach. Following the user-centered process, we will continue to refine our emphases as more scenarios become available. Whenever appropriate, we will build our scenarios on existing scenarios used in related areas, such as the ones developed for interactive systems, intelligent environment, pervasive computing and public safety communications [18-22].

References

[1] Japan Assistive Products Association, http://www.jaspa.gr.jp/, April 2003.

[2] "Global Aging," *BusinessWeek*, January 31, 2005.

[3] Jane W. S. Liu, B. Y. Wang, C. S. Shih, T. W. Kuo, A. C. Pang, C. Y. Huang, H. Y. Liao, "Reference Architecture of Intelligent Appliances for the Elderly," *Proceedings of International Conference on System Engineering*, Las Vegas, NV, August 2005.

[4] W. H. Chen, P. H. Tsai, P. C. Hsiu, C. S. Shih, Jane W. S. Liu, T. W. Kuo, A. C. Pang, and B. Y. Wang, "Specification and Compliance Enforcement of Medication Schedules," Technical Report No. TR-IIS-05-008, Institute of Information Science, Academia Sinica, Taiwan, August 2005.

[5] R. Glidden, *et al*. "Design of Ultra-low-cost UHF RFID Tags for Supply Chain Applications," *IEEE Communications*, Vol. 42, No. 8, August 2004.

[6] A. Ward, A. Jones and A. Hopper, "A New Location Technique for the Active Office," *IEEE Personal Communications*, Vol. 4, No. 5, October 1997.

[7] N. Priyantha, A. Chakraborty, and H. Balakrishnan, "The Cricket Location-Support Systems, *Proceedings of the 6th ACM MOBICOM*, August 2000.

[8] K. J. Wong, "An Ultrasonic Compass for Context-Aware Mobile Applications, M. Eng. Thesis, Massachusetts Institute of Technology, June 2004.

[9] S. Helal W. Mann, H. El-Zabadani, J. King, Y. Kaddoura, and E. Jansen, "The Gator Tech Smart House: A Programmable Pervasive Space," *IEEE Computer*, March 2005.

[10] E. Dishman, "Inventing Wellness Systems for Aging in Place," IEEE *Computer*, May 2004.

[11] A. Pentland, "Healthwear: Medical Technology Become Wearable," IEEE *Computer*, May 2004.

[12] D. A. Ross, "Cyber Crumbs for Successful Aging with Vision Loss," *IEEE Pervasive Computing*, April 2004.

[13] Changing Places Consortium, http://architecture.mit.edu/house, MIT.

[14] Aware home, http://www.cc.gatech.edu.fce/ahri/. Also FCE Smart House Research Survey, http://www.cc.gatech.edu/fce/seminar/fa98-info/smart_homes.html, Georgia Tech.

[15] Center for Future Health, http://www.futurehealth.rochester.edu/, University of Rochester.

[16] Marc smart home, http://marc.med.virginia.edu/projects_smarthomemonitor.html, University of Virginia.

[17] Assisted Cognition Projects, http://www.cs.washington.edu/assistcog/, University of Washington.

[18] K. Crisler, *et al.*, "Considering the User in the Wireless World," *IEEE Communications*, September 2004.

[19] ISO 13407, http://www.usabilitynet.org/tools/13407stds.htm, Human Centered Design Process for Interactive Systems.

[20] K. Ducatel, et al. "Scenarios for Ambient Intelligence in 2010," *IPTS-Seville*, February 2001.

[21] MIT Project Oxygen, http://oxygen.lcs.mit.edu/Overview.html

[22] Project MESA (Mobile Broadband for Emergency and Safety Applications), http://www.projectmesa.org/

Appendix Implementation Details

This appendix provides details on preliminary implementations of smart pantry described earlier in Chapter 3. The descriptions are in c-like pseudo code. As a rule, we name all global constants and variables by lower-case strings of words connected by underscores. Examples are pantry_runs and very_long_time. Each local variable or function is named by a string of one or more capitalized words.

A.1 Main Thread in Picture-Id Version of Smart Pantry

In the picture-id version of smart pantry, the image processing thread commands the camera to take pictures of the pantry periodically and processes the pictures to extract pictures of objects in it. This section describes the work done by this thread. The pantry employs one or more threads to generate and send purchase orders and to interact with the user, etc. They are straightforward to implement and are not described here.

The following global constants and variables are referred to in the pseudo code below:

- Lists: orders and error_message;
- Flags: pantry_runs = TRUE; first_time = TRUE;
- Integers: rows = 0; columns = 0;
- Array: picture[rows, columns];
- Files: compartment_config; empty; empty_pantry;
- Constants: very_long_time = 120 seconds; long_time = 60 seconds; short_time = 30 seconds; and very_short_time = 15 seconds.

The image processing thread executes the following code:

```
...
Current;                    // Current picture of the pantry
Previous;                   // Previous picture of the pantry
Configuration;              // A structure defining pantry configuration
Empty;                      // A picture of an empty compartment
CurrentContent;             // Picture of the object currently in a compartment
ItemToBeOrdered;            // Picture of an object to be ordered
Timeout = 0;

// first_time is the reading of a latch. It is TRUE when the pantry is powered on for the first
// time and the pantry is empty. This thread switches off the latch after processing the
// first picture taken of the pantry to generate compartment_config and a picture of empty
```

```
// compartments. The latch remains off until it is manually reset; this happens only when the
// physical configuration of the pantry is changed (e.g., some shelf or partition is removed.)

if (TRUE == first_time) {                    // Initialization when powered on for the first time
    Command to camera to take Current picture of the pantry;
    Save Current in file empty_pantry for later use;

    Status = GenerateCompartmentConfigurationFromPicture(&Current, &Configuration, &Empty);

    // If successful, this function produces Configuration of the pantry and a picture, Empty, of
    // compartments. It also gets the numbers of rows and columns and allocates the array
    // picture[rows, columns] and sets the elements to Empty.

    if (SUCCESSFUL == Status) {
        Save configuration in file compartment_config;
        Save picture Empty in file empty;
        Get rows and columns from Configuration;
        first_time = FALSE;
    } else
        return initialization_failed;
    }
} else {                                     // Initialization subsequent times
    Copy Configuration from compartment_config;
    Copy Empty from empty;
    Get rows and columns from Configuration;
    picture = AllocateMemory(rows * columns * sizeof(content_picture));
    if (NULL != picture)
        for every compartment (i, k), picture[i, k] = Empty;
    else return initiatization_failed;
}
Previous = Current;                          // Initialization is complete.

while (FALSE != pantry_runs) {
    WaitForPeriodEnd(long_time);       // Monitors pantry contents periodically.

    // Note that in every day use, something may block the camera from seeing parts of pantry.
    Duration = very_long_time;
    do {
        Command the camera to take Current picture of the pantry;
        Decrement Duration;
    } while (some compartment boundary is obscured && Duration > 0);
    if (0 <= Duration) {
        Send error_message “Something is blocking the camera from seeing the pantry”;
        continue;
    }
    if (Current != Previous) {
        for every compartment (i, k) in the pantry {
            CurrentContent = GetCurrentPictureOfObjectIn(i, k);
```

```
            if (CurrentContent != picture[i, k]) {          // The content of (i, k) has changed.
                if (CurrentContent == Empty) {              // A removal occurred.
                    ItemToBeOrdered = picture[i, k];
                    Generate and queue an order containing ItemToBeOrdered;
                    picture[i, k] = Empty;
                } else {
                    // The content of (i, k) may have changed. Update the picture.
                    picture[i, k] = CurrentContent;
                }
            }
        }
    }
}
// The pantry is shutting down. Clean up and exit.
```

A.2 Implementations of Bar-Code Version of Smart Pantry

This section describes ways to implement the bar-code version of smart pantry. The implementations described below make the following assumptions for the sake of clarity. (Many of these assumptions can be removed or changed without affecting the overall design.)

A.2.1 General Assumptions

Again, we assume that wired binary sensors are used to monitor whether compartments are empty or nonempty. Without loss of generality, we assume that sensor interfaces are configured as shown in Figures 3-7 and A-1. There is a sensor interface similar to the one shown here for each row of compartments (or some number of compartments). The interface requests an interrupt when any of the compartments changes from being empty to being nonempty or vice versa.

The sensors and the (pantry) system keep track of compartment states. A compartment can be in one of the following states:

- EMPTY = 0: The compartment is empty.
- NUID = 1: The compartment is not empty, but the object in it is unknown.
- NDS = 3: The bar code of the object in the compartment is known, but user preference for the kind of object is not available.
- NK = 7: The compartment is not empty, and information on its content is complete.

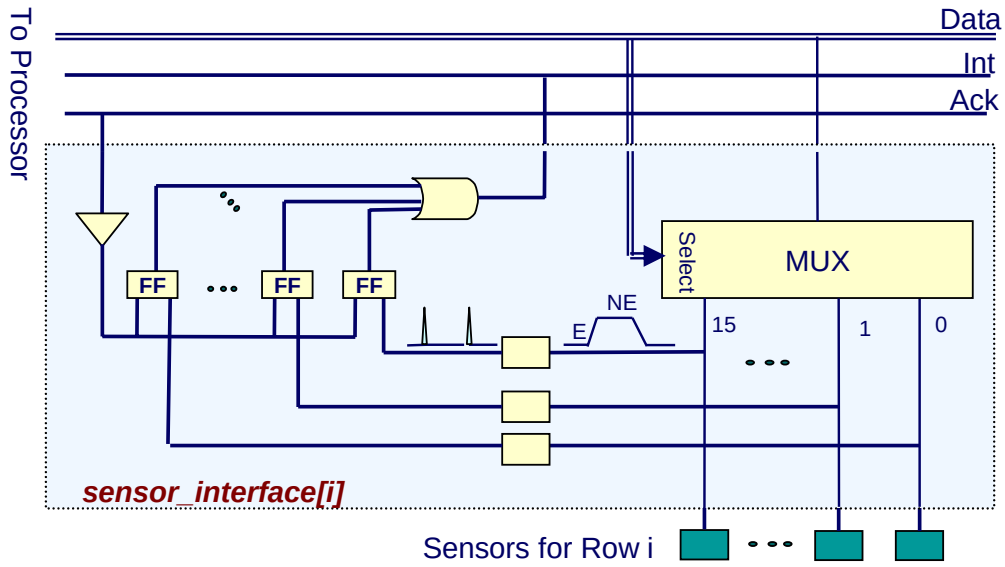

Figure A-1 Wired Sensor Interface in Bar-Code Version

Global Queues, Variables and Constants The system keeps the following FIFO queues: We assume that there is a function similar to WaitForSingleObject() of Microsoft Windows, which a thread can call to wait for entries in a specified queue and, when awaken by the presence of entries, process the entries.

- RUO_queue: a queue of RUO (removal of unknown objects) entries,
- SPO_queue: a queue of SPO (scan-and-put objects) entries,
- RKO_queue, a queue of RKO (removal of known objects) entries,
- EP_queue, a queue of error reports to be delivered to the user, and
- Order_queue, a queue of objects to be ordered from their suppliers.

Global variables and constants maintained by the pantry include the following: The names tell more or less what they are.

- Bar-code ids: SPO_wait_for_compartment_id; bar_code;
- Flags: pantry_runs = TRUE; BCA (bar_code_available) = FALSE;
- Automatic reset event: object_placed;
- Manual reset event: load_pantry;
- Array: current_state[rows, columns] = {EMPTY};
- Timeout lengths:
  very_long_time = 60 seconds;
  long_time = 30 seconds;

short_time = 10 seconds;

very_short_time = 5 seconds;

All the global variables are defined and initialized at initialization, which we omit here.

Sensor Interrupt Handler After initialization, the sensor interrupt handler executes when the state of any compartment changes from empty to nonempty and vice versus. As described by the pseudo code below, when it awakes to process an interrupt, it generates an entry for each compartment state change and inserts the entry in the corresponding queue to be processed later by a thread. (The description mentioned SPO thread: It is the thread that processes entries in the SPO_queue and handles the interaction with the user in load-pantry mode. We will return to describe the thread shortly.)

```
// Rightmost bit of current_state[i, k], PreviousReading[i, k] and CurrentReading[i, k]
// being 0 or 1 indicates that compartment (i, k) is empty or non-empty, respectively.
...
CurrentReading[I, K] = {0};                          // Array of current sensor values
PreviousReading[I, K] = {0};                         // Array of previous sensor readings

InitializeInterruptHandler();

while (pantry_runs) {
    WaitForInterrupt(FOREVER);                       // Wait without timeout

    // Interrupt occurred; poll compartments
    for every row i of compartments {
        if (sensor_interface[i] raised interrupt) {
            for every compartment (i, k) in row i {
                Get current value CurrentReading[i, k] of the sensor reading;
                if (PreviousReading[i, k] != CurrentReading[i, k]) {
                    if (0 == CurrentReading[i, k]) {        // A removal has occurred
                        if (NUID == current_state[i, k]) insert the removal into RUO_queue;
                        else insert the removal into RKO_queue;
                        current_state[i, k] = EMPTY;    // Mark the compartment empty
                    } else {
                        // A placement has occurred.
                        current_state[i, k] = NUID;     // Mark the compartment non-empty
                        if ( (load_pantry is not set) || (FALSE == BCA) ) {
                            Insert entry for (i, k) in SPO_queue;
                        } else {
                            // An object is placed in the compartment after it is scanned
                            // in the load pantry process. Signal the SPO thread.
                            Put the compartment id in SPO_wait_for_compartment_id;
                            SignalEvent(object_placed);
                        }
```

```
                        }
                    }
                    PreviousReading([i, k] = CurrentReading[i, k];
                }
                AcknowledgeInterrupt (sensor_interface[i]);      // Ready interface for interrupt
            }
        }
    }
    ...
    Disconnect();
```

Bar-Code Scanner Operation In addition, we assume that the bar code scanner operates as shown in Figure A-2. After reading a new bar code, the scanner raises an interrupt. When servicing an interrupt, the bar-code interrupt handler transfers the code to bar_code, sets the BCA (bar-code-available) flag, and acknowledges the interrupt.

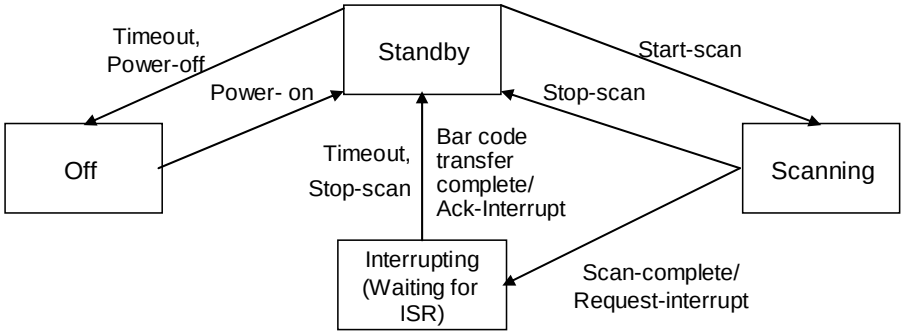

Figure A-2 Bar-Code Scanner Operations

A.2.2 Design for Responsiveness: Multi-Threaded Pantry Control

The first implementation uses five *pantry threads*, one for each of the queues described above. They are named accordingly. There are also interactive system threads that handle I/O from keyboard, touch screen and audio interface; the description below omits the system threads and concentrates on the RUO, RKO and SPO threads.

The pseudo code descriptions below make the following assumptions about threads.

- All pantry threads run on the same processor.
- RUO thread has the highest priority, SPO thread has the next highest, followed by RKO

thread, EP thread and Order thread.

- Scheduling policy is FIFO among equal priority.
- Priorities of other interactive threads relative to pantry threads are to be determined.
- When scanner interrupt handler loads a new code in bar_code, it sets the BCA (Bar Code Available) flag. When the code is read, the reading thread clears the flag.
- The audio interface serves a thread at a time for the utterance of every sentence. In other words, each sentence is stated non-preemptably. .
- Bar code read each time is returned to the last caller. So, waiters for bar code may not be awakened in priority order.

RUO Thread The RUO thread processes entries in the RUO queue as follows:

```
…
CurrentCompartment = NULL;                    //The thread is not processing any compartment
while (pantry_runs) {
    WaitForQueueObject(RUO_queue, FOREVER);       // Wait without timeout

    // RUO_Queue is no longer empty. Process the entry at the head of the queue
    CurrentCompartment = RemoveFromQueueHead(RUO_queue);
    Get compartment row i and column k from CurrentCompartment;
    Say (“You have removed the last of … Please scan its bar code);

    WaitTime = 0;                                          // Start busy wait
    while ((FALSE == BCA) and (WaitTime < short_time)) {
        Read BCA and increment WaitTime;              // Poll for 10 seconds
    }
    If (FALSE != BCA) {
        Get object Id from bar_code and clear BCA flag;
        Interact with user to confirm the order;
        if (the current object is to be reordered) {
            Generate an order for the object;
            Insert the order to be sent in the Order_queue;
        }
    } else {
        // Time expired. Object remains unidentified -- an unrecoverable error
        Generate an error report on time and compartment where the object was removed;
        Insert the error report in the EP queue.
    }
}
// pantry_runs is off. Clean up the RUO queue and exit
```

RKO Thread The RKO thread processes entries for removals of known objects as follows:

```
…
CurrentCompartment = NULL;                // The thread is not processing any compartment
```

```
Touched = FALSE;
while (pantry_runs) {
    WaitForQueueObject(RKO_queue, FOREVER);

    // RKO_queue is no longer empty. Process the entry at the head of the queue.
    CurrentCompartment = RemoveFromQueueHead(RKO_queue);
    Get compartment row i and column k from CurrentCompartment;
    Get object id from compartment descriptor;
    Ask user to touch "No" to cancel reorder;
    Touched = WaitForTouchScreen(very_short_time);
    If (FALSE == Touched) {
        //The user did not cancel the order
        Interact with the user to change supplier, delivery time, etc. as needed;
        Generate and queue reorder message;
    }
    //The user cancels the order. Do nothing for this object
}
// pantry_runs flag is off. Clean up the RKO queue and exit.
```

SPO Thread The thread that handles the scan-put objects works as follows:

```
 …
CurrentCompartment = NULL;
Placed = FALSE;
while (pantry_runs) {
    // Wait forever for an entry in SPO_queue, or load_pantry is set.
    // load_pantry event is set when the user touches "Load Pantry" or "Resume" buttons.
    WaitForMultipleObjects(SPO_Queue, load_pantry, ANY, FOREVER);

    // Give higher priority to placements of objects without Ids in SPO_queue.
    CurrentCompartment = RemoveFromQueueHead(SPO_queue);
    while (NULL != CurrentCompartment) {
        // An object is placed in compartment (i, k) while pantry is not in load pantry mode.
        Prompt the user to scan the object;
        BCA = WaitForScanner(short_time);
        if (FALSE != BCA) {
            Get object id of the compartment from bar_code and clear BCA;
            if (the object is new to the pantry) {
                current_state[i, k] = NDS;
                Prompt user for audio description;
                Add descriptor for the object to object database;
            } else {
                current_state[i, k] = NK;
            }
        } // Else do nothing. Current state was already set to NUID by sensor interrupt handler.
        CurrentCompartment = RemoveFromQueueHead(SSO_queue);
    }
```

```
// Enter load pantry mode.
while (load_panty is set)   {
        if (FALSE == BCA)   {
              Prompt user to scan object;
              BCA = WaitForScanner(short_time);
        }
        if (FALSE != BCA) {
               Get object id from bar_code;
               if (the object is new)   {
                      Prompt user to provide audio description;
                      Add descriptor of the new object;
               }
               Prompt the user to put away the object;
               Placed = WaitForEvent (object_placed, short_time);
               if (FALSE != Placed) {
                      Get compartment id (i, k) from SSO_Wait_for_Compartment_Id;
                      Placed = FALSE;
                      BCA = FALSE;
                      Update compartment state current_state(i, k) and object id;
               else {
                      // time expires. Clears BCA and get out load pantry mode
                      BCA = FALSE;
                      ResetEvent(load_panty);
               }
        } else {
               // No bar code. Get out of load pantry mode
               ResetEvent(load_pantry);
        }
    }
    // do some more work if any
}
// pantry_runs is off. Clean up and exit
```

A.2.3 Design for Simplicity: Single-Threaded Pantry Control

Rather than using three threads to process placements and removals of object, an alternative is to have one thread do all the work. That thread may be implemented as described by the pseudo code below. The description omits details on how RUO, SPO, and RKO entries are processed. The functions with named such as ProcessRUOEntries do this work in manner similar to the way RUO, SPO and RKO threads work, which was described earlier.

```
// This thread checks the RUO_queue, SPO_queue, load_pantry event, and RKO_queue
// in an order that gives removals of unknown objects the highest priority, random placements
// of objects in pantry the next highest priority, scan-put object in load pantry mode the next priority,
```

```
    // and removals of objects with known ids the lowest priority.

    CurrentCompartment = NULL;
    Placed = FALSE;
    …
    while (pantry_runs) {
        WaitForMultipleObjects(RUO_queue,
                               SPO_Queue,
                               RKO_queue,
                               load_pantry,
                               ANY,
                               FOREVER);
RepeatProcessingRUO:
        CurrentCompartment = RemoveFromQueueHead(RUO_queue);
        while (NULL != CurrentCompartment) {
            // An object of unknown id has been removed.
            ProcessRUOEntry(CurrentCompartment);        //See RUO Thread for detail.
            CurrentCompartment = RemoveFromQueueHead(RUO_queue);
        }

        // RUO_queue is empty. Go take care of random placements of objects if any.
        CurrentCompartment = RemoveFromQueueHead(SPO_queue);
        while (NULL != CurrentCompartment) {
            ProcessSPOEntry (CurrentCompartment);       // See SPO Thread for detail
            CurrentCompartment = RemoveFromQueueHead(SPO_queue);
        }

        // If the pantry is in load pantry mode, take care of one object, go back to check RUO queue.
        // Number of scan-put objects processed each time is a tunable parameter.
        while (load_panty is set)   {
            if (FALSE == BCA)   {
                Prompt user to scan object;
                BCA = WaitForScanner(short_time);
            }
            if (FALSE != BCA) {
                Get object id from bar_code;
                if (the object is new) {
                    Prompt user to provide audio description;
                    Add object descriptor;
                }
                Prompt the user to put away the object;
                Placed = WaitForEvent (object_placed, short_time);
                if (FALSE != Placed) {
                    Get compartment id (i, k) from SPO_Wait_for_Compartment_Id;
                    Placed = FALSE;
                    BCA = FALSE;
                    Update compartment state current_state[i, k] and object id;
```

```
                    goto RepeatProcessingRUO;           // go back to check RUO queue
                else {
                    // time expires. Clears BCA and get out load pantry mode
                    BCA = FALSE;
                    ResetEvent(load_panty);
                }
            } else {
                ResetEvent(load_pantry);        // No bar code. Get out of load pantry mode
            }
        }
        // Process the RKO_queue and ask whether to reorder. Process only one removal
        // of object with known id and return to check the RUO_queue again.
        CurrentCompartment = RemoveFromQueueHead(RKO_queue);
        if (NULL != CurrentCompartment) ProcessRUOEntry (CurrentCompartment);
}   // end while(pantry_runs)
// pantry_runs is off. Clean up and exit.
```