High Level Programming for Embedded Developers

Judge Maygarden
jmaygarden@ieee.org
About Me

- Firmware engineer at ActiGraph
- Past experience
  - Avionics maintenance, air-traffic control and flight simulation trainers
  - Optical and radar range tracking systems
  - Ruggedized displays and special missions avionics
- BS in Electrical and Computer Engineering from the University of Alabama
Overview

- Embedded Software Development
- Survey of Programming Paradigms
- Application of High-level Methods to Constrained Systems
- C-Language Examples
What is firmware?

- Synonymous with Embedded Software
- Usually describes fixed, small programs that are internal to electronic devices
- Traditional examples:
  - TV remote control
  - Anti-lock brakes
- Modern examples:
  - Smart phones
  - GPS navigation systems
Who writes firmware?

- Traditionally electrical engineers
- Strong grasp of hardware design
- Minimal knowledge of software construction
Embedded Software Development

- Traditional firmware
  - Small code-base focused on a specific task
  - Fixed functionality
- Modern firmware
  - Performs varied tasks with diverse interfaces
  - Configurable and field updatable
Legacy ActiGraph research devices

1. Initialize from PC
2. Sample sensors
3. Perform filtering
4. Record data
5. Download to PC
Embedded Software Development

- Future ActiGraph consumer devices:
  - Graphical display and user interface
  - Wireless connectivity
  - Meal/goal tracking
  - Music player
  - Workout modes
  - And so on...
Increased firmware responsibility requires more robust and maintainable software designs.

Imperative programming languages and constructs are still required because of memory and processing speed limitations.

Progress in high-level languages with comparatively unlimited resources is still applicable to firmware.
Survey of Programming Paradigms

- Object-Oriented Programming
- Functional Programming
- Event-driven Programming
Object-Oriented Programming

- Arrange programs into a network of systems responsible for their own data and algorithms
- Key Ideas
  - Encapsulation
  - Dynamic Dispatch and Polymorphism
Encapsulation

- Also known as Information Hiding
- Grady Brooch defines encapsulation as "the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation."
Encapsulation (cont.)

- Improves maintainability and flexibility
  - Since object internals are hidden, tight coupling between components is reduced
  - Object algorithms and internal data structures can be modified without affecting other components
  - Influences on state are isolated to the responsible object (reduces debugging time!)
Dynamic Dispatch

- Objects determine code to be executed when invoked at run-time
- Provides a more expressive replacement of large conditional structures
- Useful for implementation of finite state machines
Polymorphism

- An object of one type to appears as an object of another type through a common interface
- Allows for different objects to be used interchangeably
- New functionality may be added (i.e. plugged-in) without changing client code
Functional Programming

- Focuses on evaluation of functions instead of operations which cause changes in state
- Key Ideas
  - Referential Transparency
  - Higher-order Functions
An expression exhibits referential transparency if it can be replaced by a value with no change to the program.

Such functions avoid dependence upon external state data.

Programs avoid mutable (non-constant) data.

Referentially opaque functions have side-effects that cause maintenance problems.

Allows trivial parallelization as well as caching of function results.
Higher-order Functions

- Higher-order functions may take other functions as arguments and return functions as results.
- Improves productivity and maintenance by reducing code duplication—especially *boilerplate* code.
- The common higher-order functions **map** and **reduce** form the basis of the MapReduce software framework that makes Google tick!
Event-driven Programming

- Program flow is determined by detection and handling of events
- Key Ideas
  - Message Passing
  - Publish/Subscribe
Message Passing

- Data is conveyed through exchange of discrete packets instead of shared state
- Sending and receiving is usually asynchronous and data is copied (versus shared)
- Allows for chain of responsibility and one-to-many event handling mechanisms
Publish/Subscribe

- Receivers are not statically bound to senders, but may subscribe to published events at runtime.
- Promotes loose-coupling of components.
- Publishing modules need not have any knowledge of the usage or consequences of events as carried out by subscribers.
A pragmatic approach is required as concessions must be made for memory and/or timing limitations.

Applicable goals of high-level constructs to observe:

- Encapsulate distinct features
- Decouple disparate systems
- Minimize code duplication
Examples in C

```c
#ifndef COLLECTION_H
#define COLLECTION_H

#include "iterator.h"

typedef struct Collection Collection;

struct Collection {
    void (*initializeIterator)(Collection *, Iterator *);
};

#endif /* COLLECTION_H */

#define ITERATOR_H

#include <stdbool.h>

typedef struct Iterator Iterator;

struct Iterator {
    bool (*next)(Iterator *);
    void * (*value)(Iterator *);
    void * collection;
    void * state;
};

#endif /* ITERATOR_H */
```
#ifndef SLIST_H
#define SLIST_H

#include "collection.h"

typedef struct SList SList;
typedef struct SNode SNode;

extern Collection * slist_to_collection(SList *);
extern SList * slist_new(void);
extern void slist_delete(SList *);
extern SNode * slist_insert(SList *, void *);

#endif /* SLIST_H */
Examples in C

```c
struct SList
{
    Collection collection;
    SNode *head;
};

struct SNode
{
    SNode *next;
    void *data;
};

static bool
next(Iterator *it)
{
    const SNode *p = it->state;
    it->state = p->next;
    return NULL != it->state;
}

static void *
value(Iterator *it)
{
    const SNode *p = it->state;
    return p->data;
}

static void
initializeIterator(SList *self, Iterator *it)
{
    it->next = next;
    it->value = value;
    it->collection = self;
    it->state = self->head;
}

Collection *
slist_to_collection(SList *self)
{
    return &self->collection;
}

SList *
slist_new(void)
{
    SList *self;

    self = malloc(sizeof (SList));
    if (self) {
        self->collection.initializeIterator =
            (void (*)(Collection *, Iterator *))
            initializeIterator;
        self->head = NULL;
    }

    return self;
}
```
void slist_delete(SList *self)
{
    SNode *p;

    while (self->head) {
        p = self->head;
        self->head = self->head->next;
        free(p);
    }

    free(self);
}

SNode *
slist_insert(SList *self, void *data)
{
    SNode *p;

    p = malloc(sizeof (SNode));
    if (p) {
        p->next = self->head;
        p->data = data;
        self->head = p;
    }

    return p;
}
Examples in C

```c
void
map(Collection *c, void (*function)(void *value, void *upvalue), void *upvalue)
{
    Iterator it;

    c->initializeIterator(c, &it);
    do {
        function(it.value(&it), upvalue);
    } while (it.next(&it));
}

void
map2(Collection *c1, Collection *c2,
     void (*function)(void *value1, void *value2, void *upvalue), void *upvalue)
{
    Iterator it1, it2;

    c1->initializeIterator(c1, &it1);
    c2->initializeIterator(c2, &it2);
    do {
        function(it1.value(&it1), it2.value(&it2), upvalue);
    } while (it1.next(&it1) && it2.next(&it2));
}
```
void *
reduce(Collection *c, void *initialValue,
       void * (*function)(void *result, void *value, void *upvalue),
       void *upvalue)
{
    Iterator it;
    void *result;
    result = initialValue;
    c->initializeIterator(c, &it);
    do {
        result = function(result, it.value(&it), upvalue);
    } while (it.next(&it));

    return result;
}
#include <stdio.h>
#include "array.h"
#include "map.h"
#include "reduce.h"
#include "slist.h"

static void
print_number(void *value, void *upvalue)
{
    printf("%s: %d\n", (char const *) upvalue, (int) value);
}

static void
print_number2(void *value1, void *value2, void *arg)
{
    printf("%s: %d, %d\n", (char const *) upvalue, (int) value1, (int) value2);
}

static void *
sum(void *result, void *value, void *upvalue)
{
    return (void *) ((int) result + (int) value);
}
int main(int argc, char *argv[])
{
    Array *array;
    SList *slist;
    Collection *c1, *c2;
    int i;

    array = array_new(10);
    slist = slist_new();
    c1 = slist_to_collection(slist);
    c2 = array_to_collection(array);
    for (i = 0; i < 10; ++i) {
        array_set(array, i, (void *) i);
        slist_insert(slist, (void *) i);
    }

    map(c2, print_number, "map");

    map2(c1, c2, print_number2, "map2");

    printf("reduce: %d\n", (int) reduce(c2, (void *) 0, sum, NULL));

    slist_delete(slist);
    array_delete(array);
    return 0;
}
$ ./ieee
map: 0
map: 1
map: 2
map: 3
map: 4
map: 5
map: 6
map: 7
map: 8
map: 9
map2: 9, 0
map2: 8, 1
map2: 7, 2
map2: 6, 3
map2: 5, 4
map2: 4, 5
map2: 3, 6
map2: 2, 7
map2: 1, 8
map2: 0, 9
reduce: 45
Questions?