EE382N : Embedded System Design and Modeling
Lab #1
Names: Behzad Boroujerdian, Kishore Punniyamurthy, Keum San Chun

Problem (a)

1. Analyze the source code structure and draw a high-level block diagram of the function hierarchy and their communication dependencies (critical variables). **Submit the block diagram of the software architecture of the reference code as part of your lab report.** Note: in addition to the edge detection, the original source doe also supports image corner detection and smoothing. You can ignore functions that are not used in the edge detection code.

   Block diagram attached at the end of this report (Reference 1).

i) Critical variables were identified to be the following:
   in, mid, bp, r, x_size, y_size, bt, drawing_mode, max_no_corners, max_no_edges, mode

2. Modify the source code to be static and synthesizable. Modify the sources for a fixed input image sensor size of 76x95 pixels. Remove any unnecessary communication/dependencies. At the end, put each function called inside the main() function in a separate header/source file in order to make the example easier. **Report on the code changes that you performed.**

i) Modified block diagram attached at the end of the report (Reference 2).

ii) The functions that are only relevant to edge detection were retained.
   The followings are the changes made:
   1) in, mid, r were arrays of specific sizes. They were originally dynamically allocated by malloc(), we changed them to arrays of static sizes based on the input size.
   2) x_size and y_size were converted as constant parameters
   3) switch block was removed and replaced with a single case which corresponds to the edge detection.

3. Make sure that parameters passed between functions are not of pointer type.

   Actual arrays and constant values were used as parameters instead of pointers. The complete arrays were sent using port function calls of specC.

EE382N : Embedded System Design and Modeling
Lab #1
Names: Behzad Boroujerdian, Kishore Punniyamurthy, Keum San Chun

Problem (b)

1 & 2 & 3) Introduce a single behavior of appropriate name in each file. Let the behavior encapsulate all local variables and functions. Replace parameters with equivalent behavior ports for external communication. Make sure that parameters passed between functions are not of pointer type.

The following files were created:

1. stimulus.sc – this file contains the behavior which replaces the get_image() C function. It reads the image file and sends the image array to other behavior. It also sends a start signal to initiate the whole execution.

2. detect_edges.sc – this file contains the behavior setupbrightness and susanedges within it. The setupbrightness creates the array "bp" which is then passed to susanedges behavior using a queue. The susanedges behavior takes input from setupbrightness behavior and generates the "mid" and "r" arrays.

3. susan_thin.sc – this file contains the behavior which performs the susan_thin() C functions, it receives inputs from detect_edges and outputs "mid" to edge_draw through queue.

4. edge_draw.sc – This behavior corresponds to the edge_draw() C function. It receives the "mid' array from susan_thin and image from the stimulus behavior and outputs the result to monitor behavior.

5. monitor.sc – this behavior corresponds to the put_image() C function. It receives the image array from edge_draw through a queue and outputs it into a file.

6. main.sc – it is the behaviour where all the modules and ports/channels are instantiated. The channels instantiated are connected between appropriate behavior.

7. constant.hh - This files contains all the parameters required in other behaviors.

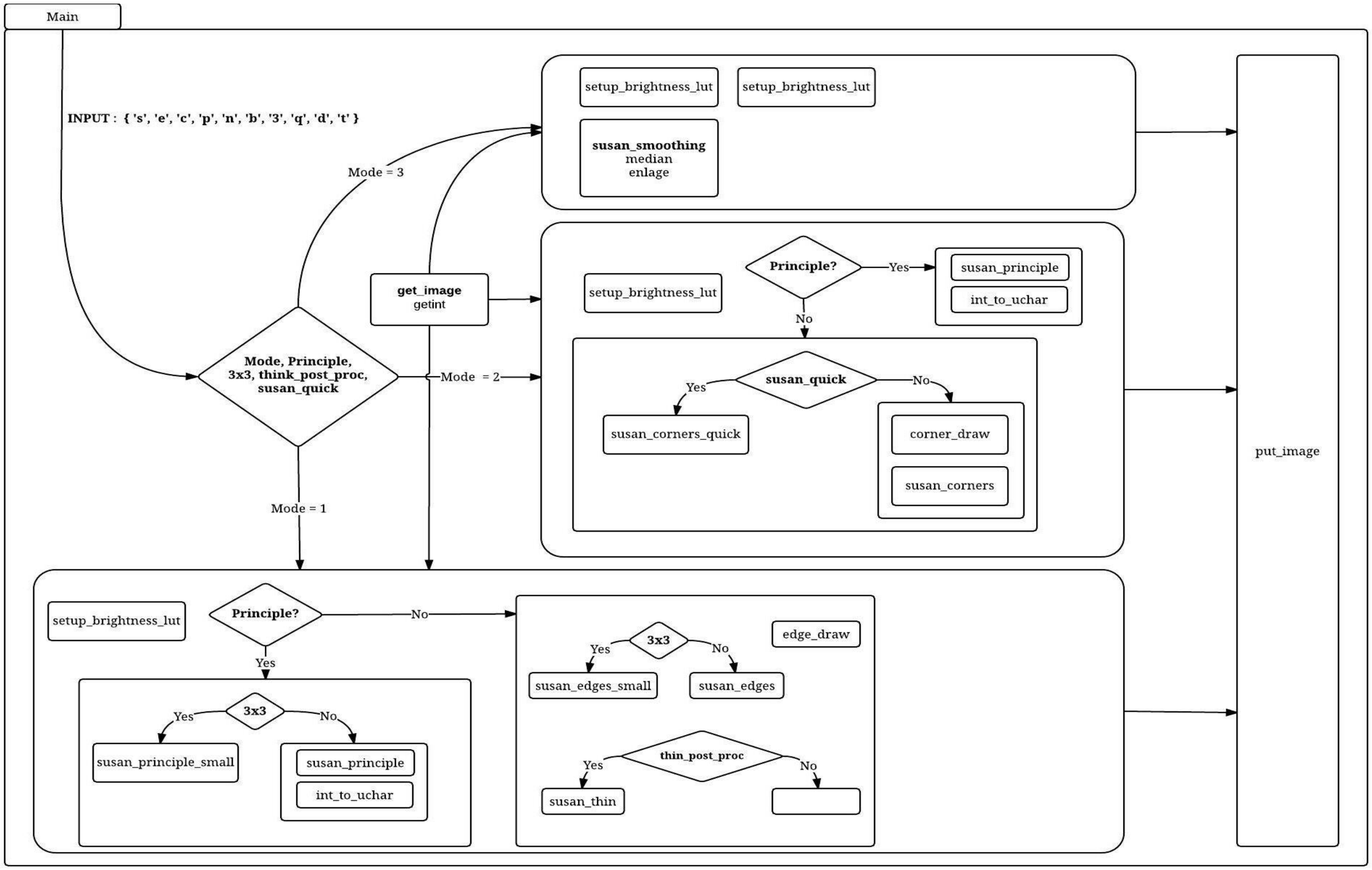
The top level intantiation:

```
c_double_handshake Trigger;

c_queue imageBuffer(img_size);

c_queue imageBuffer2(img_size);

c_queue detectEdgeOutputR(img_size4);

c_queue detectEdgeOutputMid (img_size);

c_queue susanThinOutput(img_size);

c_queue edgeDrawOutput(img_size);

stimulus myStimulus(imageBuffer, Trigger, imageBuffer2);
```

```
detectedges mydetectEdges(imageBuffer, Trigger, detectEdgeOutputR,
detectEdgeOutputMid);

susan_thin mySusanThin(detectEdgeOutputR,
detectEdgeOutputMid,susanThinOutput);

edge_draw myEdgeDraw(susanThinOutput, imageBuffer2, edgeDrawOutput);

monitor myMonitor(edgeDrawOutput);
```

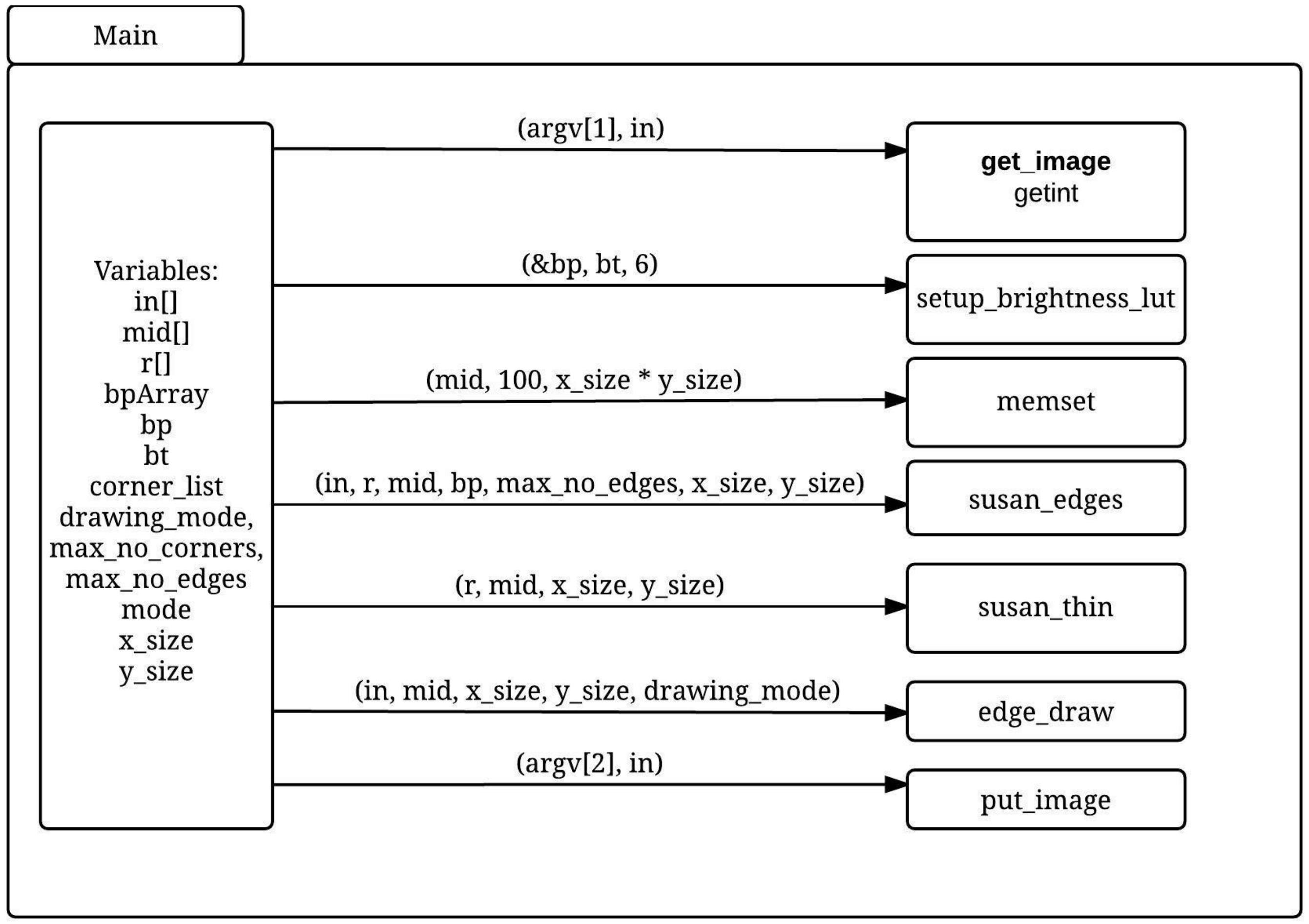
EE382N : Embedded System Design and Modeling
Lab #1
Names: Behzad Boroujerdian, Kishore Punniyamurthy, Keum San Chun

Reference 1]Block diagram of the original version of susan.c

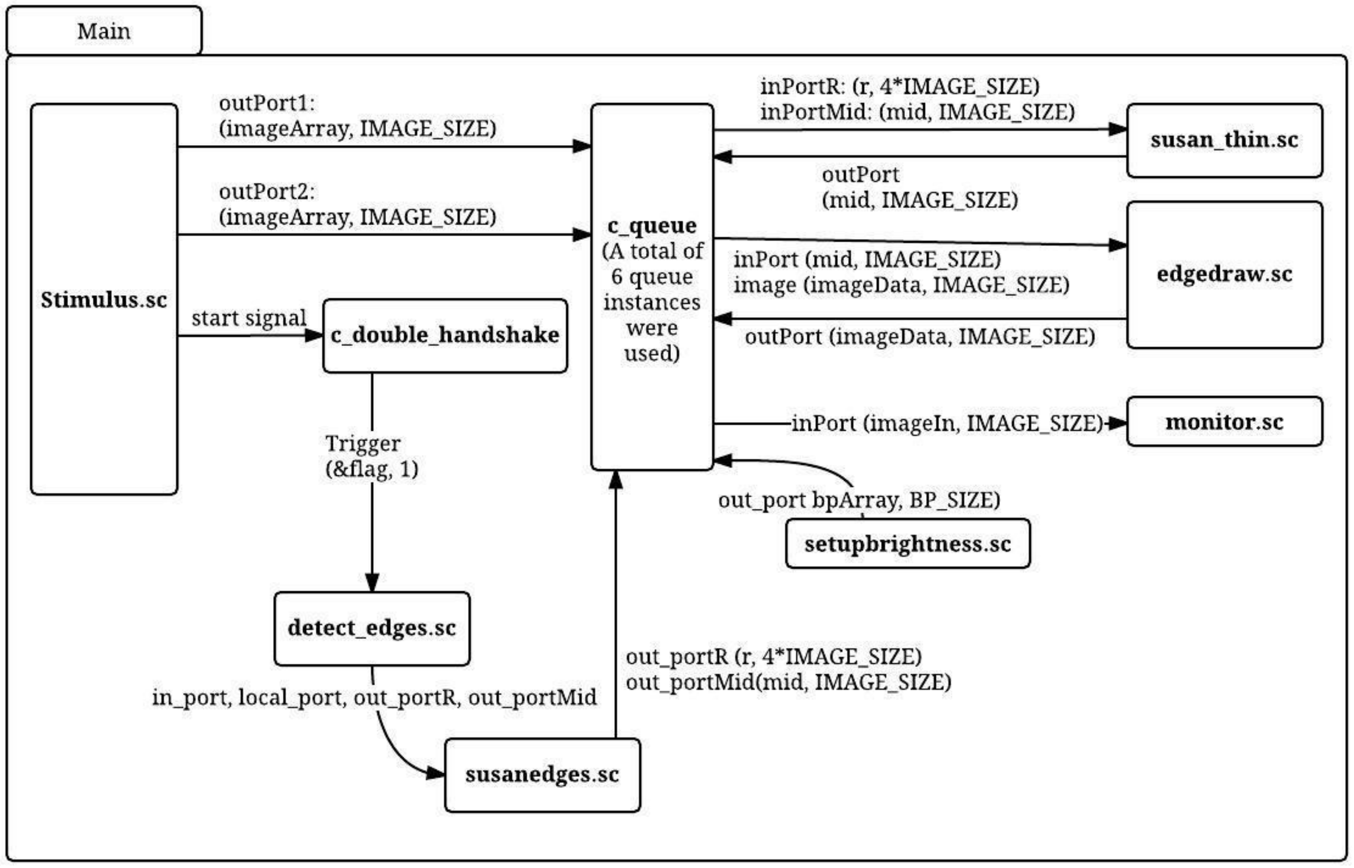


EE382N : Embedded System Design and Modeling
Lab #1
Names: Behzad Boroujerdian, Kishore Punniyamurthy, Keum San Chun

[Reference 2]Block diagram of the simplified version of susan.c



[Reference 3]Block diagram of main.sc


Main
Stimulus.sc
outPort1: (imageArray, IMAGE_SIZE)
outPort2: (imageArray, IMAGE_SIZE)
start signal
c_double_handshake
Trigger (&flag, 1)
detect_edges.sc
in_port, local_port, out_portR, out_portMid
susanedges.sc
out_portR (r, 4*IMAGE_SIZE)
out_portMid(mid, IMAGE_SIZE)
c_queue (A total of 6 queue instances were used)
inPortR: (r, 4*IMAGE_SIZE)
inPortMid: (mid, IMAGE_SIZE)
susan_thin.sc
outPort (mid, IMAGE_SIZE)
inPort (mid, IMAGE_SIZE)
image (imageData, IMAGE_SIZE)
edgedraw.sc
outPort (imageData, IMAGE_SIZE)
inPort (imageIn, IMAGE_SIZE)
monitor.sc
out_port bpArray, BP_SIZE)
setupbrightness.sc

EE382N : Embedded System Design and Modeling
Lab #1

Names: Behzad Boroujerdian, Kishore Punniyamurthy, Keum San Chun

Part C.3

Memory Boundedness

Our model can run in bounded memory provided, each behavior runs at similar rate, if not then, it cannot run in bounded memory. For e.g, stimulus runs at a very fast rate but susan runs very slow, then as time progress, the size of queue grows larger and larger (no bound) . The model can execute with queue size = image size. Limiting the queue size affects the scheduling by bring an order (corresponding to data flow) to execution.

Queue size dependency

Depending on the queue size between behaviors, we can utilize parallelism.We consider three scenarios for queue size:If the size of the queue between behaviors is greater than the imageSize, then connected behaviors can run in parallel. However, if the queue size is equal to the imageSize, the flow will be done in sequential manner as shown below:

Stimulus → read_image → susan → write_image → monitor

If the queue size is less than the imageSize, the program will not work (deadlock)

Queue size Justification:

As long as all the behaviors are working as they are meant to be, and queue size is at least 1 image size, the model is deadlock free and it is deterministic, as there is no shared resource.

D.1

We achieve the following timing number from sequential and parallel running(using –par) of our code:

```
sequential: 0.45user 1.19system 0:01.80elapsed 91%CPU (0avgtext+0avgdata 115200maxresident
parallell: 0.45user 1.13system 0:01.75elapsed 90%CPU (0avgtext+0avgdata 115728maxresident)
```

As can be seen, we see a slight improvement on the system time when run in parallel. We believe that this is due to the fact that running our code in multicore, we are capable of running the behaviors in parallel

Part D.2

We chose to parallelize edge draw behavior. In our implementation , we are instantiating 2 instances of edge_draw behavior which processes half image parallely. Edge_draw requires 2 inputs – mid and input_image. We created separate behavior for splitting the mid and image into 2 sections which can be sent each of the edge_draw instance. Within edge_draw, processing each image element modifies few image elements before and after the specific element in consideration. We split the image into 2 parts with overlapping sections, the image is 7220 (IMG_SIZE) elements long, 1 part is image [0: (IMG_SIZE/2 + offset)] and 2nd part is image [(IMG_SIZE/2 – offset) : IMG_SIZE-1], where offset is 133 (decided based on input and dependencies. The processed outputs from each edge_draw instance is then merged together using another newly created behavior. (please refer to the png pictures provided)

This model implemented as KPN reveals the task level parallelism but the parallelism within a task is not explicitly highlighted. There is potentially parallelism to be exploited in susan_thin and susan_Edge behavior which is unknown until implemented parallelly (or expressed in a different computational model where the tool can be guided to synthesize the design in a parallel fashion)
Implementing it using different MOC (like Data flow graph with finer granularity) would show the parallelism within task more explicitly.

Yes, this model(our code) can be expressed using SDF

could a tool be developed to recognize if a model is KPN or SDF ?
- if a program terminates without deadlocking we can claim it is definitely not a KPN(since KPN termination point is an artificial deadlock)

- if there exists a loop in the model, then it is definitely not a KPN.

Advantage KPN over SDF:

Turing complete

Deterministic

Advantage SDF over KPN:

Memory bounded analysis possible

Static schedule of actors

Deadlock analysis

EE382N : Embedded System Design and Modeling
Lab #1
Names: Behzad Boroujerdian, Kishore Punniyamurthy, Keum San Chun

Note: all of our images (regarding the questions above) are stored in a folder called “images”