Lab 3 – Basic C programming

Lab targets: writing basic C programs using header files and conditional compilation, program debugging using GDB, assert(), and a debug mode, usage of keywords const, static and extern.

Exercise: *typedef, functions, multiple files, conditional compilation*

THE SOLUTION OF THIS EXERCISE NEEDS TO BE UPLOADED ON TOLEDO BEFORE THE NEXT LAB.

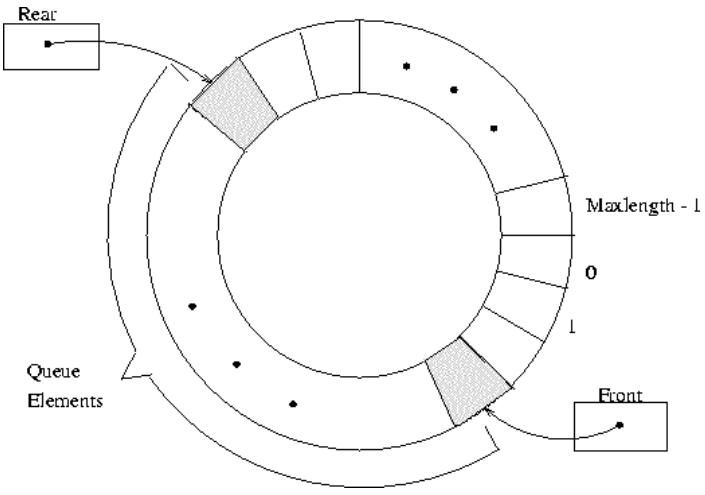
A queue is a data structure that implements the 'first come, first serve' principle. A queue is also called a FIFO (First in, first out). A queue is an important data structure in many Operating Systems tasks, e.g. in multi-tasking management, print queues, buffering queues (e.g. socket buffers, disk buffers,…), priority queues, etc. Write a C program that implements and tests a queue. Use at least 3 files: main.c, queue.c, and queue.h. The queue.c file contains the queue variable to access the queue. Hint: look at the 'stack' implementation used in the lectures as an example and starting point. Use a **circular array** as underlying data structure to store the queue elements. At least the following operators should be supported:

- QueueCreate: creates and initializes the queue and prepares it for usage
- QueueDestroy: deletes the queue from memor; the queue can no longer be used unless QueueCreate is called again
- QueueSize: return the number of elements in the queue
- QuequeTop: return the top element in the queue
- Enqueue: add an element to the queue
- Dequeue: remove the top element from the queue

Use **storage class** specifiers (extern, static, etc.) where needed. For example, to protect access to the queue variable and local functions in queue.c from other files like main.c.

Use conditional compilation to allow a user to run the code in 'debug mode'.

Use Assert(…) to catch exceptions in your code that should never happen.



Rear is the position in the queue where new elements are added (enqueue).

Front is the position in the queue where elements are removed (top, dequeue).

Exercise: *GDB*

Compile your code with debug information (How? Check man-pages of gcc.). Run gdb to inspect variables, set breakpoints and use the step-by-step execution mode.

Exercise: *preprocessor, conditional compilation, gcc toolchain*

Change the queue implementation of the previous exercise such that at compile time the user can set the maximum size of the queue and can choose the data type of the elements of the queue. If the user doesn't define a size and/or a data type at compile time, a default size and data type should be chosen. Only atomic data types can be chosen by the user, i.e. short, int, long, float, double, and char. You may assume that the user will enter its choice in capital letters, e.g. INT or DOUBLE.

Exercise: *pre-processor*

Run the pre-processor (How? Use the man-pages of gcc!) on the code of the previous exercise and save the result in a text file. Open this text file and find out what the pre-processor has done with (i) #include, (ii) #define, and (iii) #ifdef statements.

Exercise: *typedef, functions, multiple files, conditional compilation*

Design and implement a basic finite state machine (FSM) engine. In the fsm.h file, the user defines the number of states and the number of labels. The labels are used to identify transitions from one state to another state. The user may also give names to every state and every label. E.g.:

Enum states {STANDBYE, WAIT, RUN };

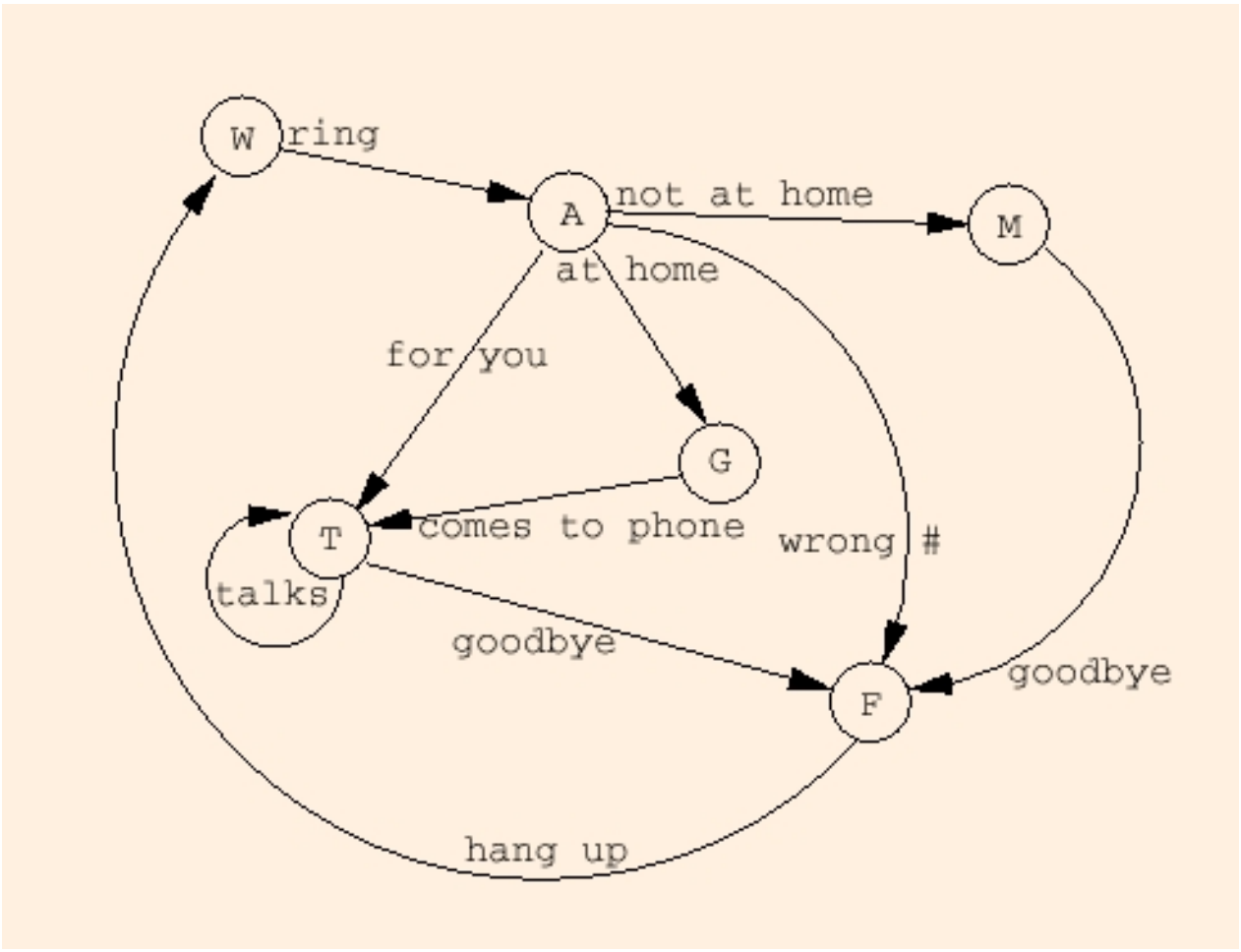
Enum labels {play, record, pause}

The fsm.c file contains the implementation of the FSM engine. At least the functions given below should be available to the user of the FSM engine. The first 4 functions are used to define/configure a new FSM, the remaining 3 functions are for using the FSM.

- InitFSM(): initialize (prepare for usage) the FSM engine; this function can also be called to reset the FSM to its original state;
- SetStartState(state): there should be only one start state;
- SetExitState(state): by calling this function multiple times, the user can set more than one exit state;
- SetTransition(state1, label, state2): defines a transition from one state to another;
- StartFSM(): start using the FSM;
- DoTransition(label): executes the transition with 'label' from the current state; if there is no such transition, an error is returned;
- IsInExitState(): check if the current state is an exit state;

Use conditional compilation to allow a user to run the FSM-code in 'debug mode'.

Of course, also write a program to test the FSM engine implementation. You can use the following state diagram as an example. The state diagram indicates what happens when you are at home and there is an incoming phone call. The state 'W' is start and exit state at the same time.


W
ring
A
not at home
M
at home
for you
G
T
comes to phone
wrong #
talks
goodbye
goodbye
F
hang up