libHLS (v2.1.0): Quick Reference Guide

Shantanu Tarafdar/Valerie Ohm **October 13, 1998**

Generic Types

Type: `Boolean` **Include file:** `#include<HLS/generic.h>`

Description: Boolean. Has two values: `FALSE` (= 0) and `TRUE` (= 1).

Type: `List` **Include file:** `#include<HLS/generic.h>`

Description: Doubly-linked list data structure. A list is made of list elements. Each holds a pointer to the real data and pointers to neighboring list elements. The type of data held by list items is `void *`. The data can be typecast to the appropriate type. The list maintains pointers to the current item, the head (first) item, and the tail.

`List *l_new()`
Creates a new list.
`void l_free(List *l))`
Frees a list. Does not free items.
`void l_rwd(List *l)`
Moves current item pointer to head of list.
`void l_fwd(List *l)`
Moves current item pointer to tail of list.
`void l_nxt(List *l)`
Moves current item pointer to next item of list from current position.
`void l_prv(List *l)`
Moves current item pointer to previous item of list from current position
`void l_ins(List *l, void *data)`
Inserts data at current position in list. Old current item moves towards tail.
`void l_prepend(List *l, void *data)`
Inserts data at head of list.
`void l_append(List *l, void *data)`
Inserts data at tail of list.
`void l_rm(List *l)`
Removes item at current position in list
`void l_del(List *l, void *data)`
Removes item whose pointer matches `data`'s from the list. Preserves current position.
`void *l_get(List *l)`
Returns the item at the current position in the list.
`void *l_first(List *l)`
Returns the first item in list.
`void *l_last(List *l)`
Returns the last item in list.
`void *l_behead(List *l)`
Removes and returns the first item in list.
`void *l_betail(List *l)`
Removes and returns the last item in list.
`Boolean l_find(List *l, void *data)`
Returns a `TRUE` if `data` is in the list. `FALSE` otherwise.
`Boolean l_lookfor(List *l, void *data)`
Returns a `TRUE` if `data` is in the list. `FALSE` otherwise. Preserves current position.
`int l_size(List *l)`
Returns number of items in list.
`Boolean l_isempty(List *l)`
Returns `TRUE` if list has no items. `FALSE` otherwise.
`Boolean l_atend(List *l)`
Returns `TRUE` if pointer was moved past either end of list. `FALSE` otherwise.
`List *l_copy(List *l)`
Makes copy of list.
`void l_sort(List *l, int (*cmpfnc) (void *data1, void *data2))`
Sorts the list with "largest" item first.
`void l_empty(List *l)`
Empties the contents of list without freeing it.
`void l_concat(List *l1, List *l2)`
Appends all items in `l2` to `l1`.
`void l_merge(List *l1, List *l2)`
Appends all items in `l2` not already in `l1` to `l1`.
`void l_reverse(List *l)`
Reverses the order of items in list.
`void l_randomize(List *l)`
Randomizes the order of items in list.
`List *l_intersect(List *l1, List *l2)`
Returns a new list containing all items common to l1 and l2.
`void l_fforeach(List *l, void *item) { ... } l_endfor;`
Fast looping construct. Current item being traversed is placed in `item`. Not safe to delete items within loop. Fast.
`void l_sforeach(List *l, void *item) { ... } l_endfor;`
Safe looping construct. Current item being traversed is placed in `item`. Safe to delete items within loop. Slow. `break`'ing ok. Do not `return` out of loop.

Type: `HashTable` **Include file:** `#include<HLS/generic.h>`

Description: Hashtable keyed with strings. Data in table is of type `void *`, but may be typecast. Dynamic resizing is available as an option.

`HashTable *ht_new()`
Creates new hashtable with size 32 and auto-resizing on.
`void ht_free(HashTable *h)`
Frees hashtable. Does not free items.
`void ht_destroy(HashTable *h, void (*freeFunc)(void *))`
Frees hashtable and applies `freeFunc()` on all items.
`void *ht_ins(HashTable *h, const char *key, void *item)`
Inserts item, keyed by string, into hashtable. Returns any previous item with same key and replaces it.
`void *ht_get(HashTable *h, const char *key)`
Returns item corresponding to key and `NULL` if none exists.
`void *ht_del(HashTable *h, const char *key)`
Removes item corresponding to key and returns it.
`void ht_empty(HashTable *h)`
Empties hashtable of all entries without freeing it.
`void ht_purge(HashTable *h, void (*freeFunc)(void *))`
Same as `ht_empty()` except `freeFunc()` is applied to all items.
`void ht_iterate(HashTable *h, void (*func)(char *, void *))`
Iterates over all items and applies `func()` to each.
`HashTable *ht_copy(HashTable *h)`
Creates a copy of hashtable.
`void ht_resize(HashTable *h, int size)`
Resizes a hashtable. This sets the number of slots in the hashtable for items to be placed in. Better performance is attained is number of items in hashtable is approx. equal to a low multiple of the number of slots.
`void ht_autoresizing_on(HashTable *h)`
Enables auto-resizing of hashtable. This keeps the multiple mentioned above less than 2.
`void ht_autoresizing_off(HashTable *h)`
Disables auto-resizing for hashtable.
`int ht_num_entries(HashTable *h)`
Returns the number of items in the hashtable.

`Boolean ht_isempty(HashTable *h)`
Returns `TRUE` if hashtable is empty and `FALSE` otherwise.
`void ht_fforeach(HashTable *h, char *key, void *item) { ... } ht_endfor;`
Fast looping construct. Current (key, item) pair being traversed is placed in `key` and `item`. Not safe to delete items within loop. Fast.
`void ht_sforeach(HashTable *h, char *key, void *item) { ... } ht_endfor;`
Safe looping construct. Current (key, item) pair being traversed is placed in `key` and `item`. Safe to delete items within loop. Slow. `break`'ing ok. Do not `return` out of loop.

Graphs

Type: `Graphs` **Include file:** `#include<HLS/graphlib.h>`

Description: Graphs are mixed directed/undirected graphs. The representation is an similar to an adjacency list. The graph has a list of nodes and one of edges. Each node has lists of incoming, outgoing, and undirected edges. Each directed edge has a source and destination. Each undirected edge has two endpoints. A graph and each node and edge can have attributes, properties, and user information. An attribute is named and has value which is a string. A property is named as well and has a user-defined data structure as value. User information is a user-defined data structure. User information can be accessed faster than properties or attributes, but there can store only one data structure. A graph also has an inports list and an outports list. These are ordered lists and are meant for use in hierarchical graphs.

`Graph *graph_new(const char *name)`
Creates new graph.
`void graph_free(Graph *g)`
Frees graph and constituent data structures.
`void graph_addinport(Graph *g, Node *n)`
Adds node to the inports list.
`void graph_rminport(Graph *g, Node *n)`
Removes node from inports list.
`void graph_remove_inport(Graph *g, const char *nodename)`
Removes named node from inport list.
`void graph_addoutport(Graph *g, Node *n)`
Adds node to the outport list.
`void graph_rmoutport(Graph *g, Node *n)`
Removes node from outports list.
`void graph_remove_outport(Graph *g, const char *nodename)`
Removes named node from outports list.
`void graph_autoaddports(Graph *g)`
Adds nodes with no incoming edges to graph inports list and those with no outgoing edges to outports list.
`Node *graph_create_node(Graph *g, const char *nodename)`
Creates node and adds it to graph.
`void graph_rmnode(Graph *g, Node *n)`
Removes node from graph.
`void graph_remove_node(Graph *g, const char *nodename)`
Removes named node from graph.
`Edge *graph_add_edge(Graph *g, Node *src_n, Node *dst_n)`
Creates edge from `src_n` to `dst_n` and adds it to graph.
`Edge *graph_add_named_edge(Graph *g, const char *edgename, Node *src_n, Node *dst_n)`
Creates edge with given name from `src_n` to `dst_n` and adds it to graph
`Edge *graph_create_edge(Graph *g, const char *srcnodename, const char *dstnodename)`
Creates edge from named node tt srcnodename to named node `dstnodename` and adds it to graph.
`Edge *graph_create_named_edge(Graph *g, const char *edgename, const char *srcnodename, const char *dstnodename)`
Creates edge with given name from named node tt srcnodename to named node `dstnodename` and adds it to graph.
`void graph_rmedge(Graph *g, Edge *e)`
Removes edge from graph.
`UEdge *graph_add_uedge(Graph *g, Node *n1, Node *n2)`
Creates undirected edge between `n1` and `n2` and adds to graph.
`UEdge *graph_add_named_uedge(Graph *g, const char *uedgename, Node *n1, Node *n2)`
Creates undirected edge with given name between `n1` and `n2` and adds to graph.
`UEdge *graph_create_uedge(Graph *g, const char *nodename1, const char *nodename2)`
Creates undirected edge between named node `nodename1` and named node `nodename2` and adds to graph.
`UEdge *graph_create_named_uedge(Graph *g, const char *ue_name, const char *nname1, const char *nname2)`
Creates undirected edge with given name between named node `nname1` and named node `nname2` and adds to graph.
`void graph_rmuedge(Graph *g, UEdge *e)`
Removes undirected edge from graph.
`char *graph_name(Graph *g)`
Returns graph name. (private)
`List *graph_inports(Graph* g)`
Returns list of inports of graph. (private)
`List *graph_outports(Graph* g)`
Returns list of outports of graph. (private)
`List *graph_nodes(Graph *g)`
Returns list of nodes of graph. (private)
`List *graph_edges(Graph *g)`
Returns list of directed edges of graph. (private)
`List *graph_uedges(Graph *g)`
Returns list of undirected edges of graph. (private)
`void graph_sforeach_attr(Graph *g, char *attr_name, char *attr_val) { ... } graph_endfor_attr;`
Looping construct. Iterates over all attributes of graph. Attribute name (private) and value (private) are placed in `attr_name` and `attr_val`. Safe to delete attributes inside loop. Slow. `break`'ing ok. Do not `return` out of loop.
`Attr_Table *graph_attrs(Graph *g)`
Returns list of graph attributes.
`void graph_sforeach_prop(Graph *g, char *prop_name, void *prop_val) { ... } graph_endfor_prop;`
Looping construct. Iterates over all properties of graph. Property name (private) and value are placed in `prop_name` and `prop_val`. Safe to delete properties inside loop. Slow. `break`'ing ok. Do not `return` out of loop.
`Node *graph_getnode_byname(Graph *g, const char *nodename)`
Returns named node of graph.
`Edge *graph_getedge_byname(Graph *g, const char *edgename)`
Returns named directed edge of graph. `NULL` if it doesn't exist.
`Edge *graph_getedge_byendpoints(Graph *g, Node *src_n, Node *dst_n)`
Returns directed edge between specified nodes. `NULL` if it doesn't exist.
`Edge *graph_getedge_bynamedendpoints(Graph *g, const char *srcnodename, const char *dstnodename)`
Returns directed edge between named nodes. `NULL` if it doesn't exist.
`UEdge *graph_getuedge_byname(Graph *g, const char *uedgename)`
Returns named undirected edge of graph. `NULL` if it doesn't exist.
`UEdge *graph_getuedge_byendpoints(Graph *g, Node *n1, Node *n2)`
Returns undirected edge between specified nodes. `NULL` if it doesn't exist.
`UEdge *graph_getuedge_bynamedendpoints(Graph *g, const char *nodename1, const char *nodename2)`
Returns undirected edge between named nodes. `NULL` if it doesn't exist.
`void graph_setattr(Graph *g, const char *attr, const char *val`
Sets the value of a graph attribute.
`char *graph_getattr(Graph *g, const char *attr)`
Retrieves the value (private) of a graph attribute.

`void graph_delattr(Graph *g, const char *attr)`
Deletes an attribute of graph.
`void graph_setprop(Graph *g, const char *prop_name, void *data, void (*freeFunc) (void *))`
Sets data structure for property of graph. Applies `freeFunc()` to data structure when property is deleted if `freeFunc != NULL`.
`void *graph_getprop(Graph *g, const char *prop_name`
Gets data structure for property of graph.
`void graph_delprop(Graph *g, const char *prop_name)`
Deletes a graph property.
`void graph_set_user_info(Graph *g, void *info)`
Sets the user information data structure
`void *graph_get_user_info(Graph *g)`
Gets the user information data structure.
`void graph_clear_user_info(Graph *g)`
Clears the user information data structure.
`void graph_print(Graph *g)`
Prints a representation of the graph to `stdout`.
`void graph_printnodes(Graph *g)`
Prints the node list of graph to `stdout`.
`void graph_printedges(Graph *g)`
Prints the directed edge list of graph to `stdout`.
`void graph_printuedges(Graph *g)`
Prints the undirected edge list of graph to `stdout`.
`void graph_write(FILE *fp, Graph *g)`
Writes the graph to a stream in libHLS graph format.
`void graph_write_to_file(char *fname, Graph *g`
Creates a file and writes the graph to it in libHLS graph format.
`Graph *graph_read(FILE *fp)`
Reads a graph from a stream assuming libHLS graph format.
`Graph *graph_read_from_file(char *fname)`
Opens a file and reads a graph from it assuming libHLS graph format.
`void graph_display(Graph *g)`
Displays graph using xvcg.
`void graph_undisplay(Graph *g)`
Closes xvcg window displaying graph.
`void graph_compute_distance_matrix(Graph *g)`
Computes the distance matrix for the graph. It is a matrix whose elements correspond to minimum and maximum distances between pairs of nodes of graph through directed edges. Distances are the number of directed edges.
`void graph_print_distance_matrix(Graph *g)`
Prints the distance matrix of the graph to `stdout`.
`int graph_get_min_distance(Graph *g, Node *src_n, Node *dst_n)`
Returns the minimum distance traversing directed edges from `src_n` to `dst_n`.
`int graph_get_max_distance(Graph *g, Node *src_n, Node *dst_n)`
eturns the maximum distance traversing directed edges from `src_n` to `dst_n`.
`Graph *graph_copy_topology(Graph *g, char *new_graph_name)`
Creates a copy of the graph. Only topology is copied. That is nodes, edges, uedges, inports, and outports. Attributes, properties, and user information is not copied.
`void graph_randomize(Graph *g)`
Randomizes all the lists associated with the graph, i.e. nodes, edge, uedges, inports, and outports.
`Boolean graph_verify(Graph *g)`
Verifies that the graph is consistent and corrects errors where it can. (Refer to manual for more information)
`Boolean graph_set_edge_src(Graph *g, Edge *e, Node *new_src_n)`
Adjusts the source node of an existing directed edge.
`Boolean graph_set_edge_dst(Graph *g, Edge *e, Node *new_dst_n)`
Adjusts the destination node of an existing directed edge.
`Boolean graph_set_uedge_n1(Graph *g, UEdge *ue, Node *new_n1)`
Adjusts the first endpoint of an existing undirected edge.
`Boolean graph_set_uedge_n2(Graph *g, UEdge *ue, Node *new_n2)`
Adjusts the second endpoint of an existing undirected edge.
`List *graph_topological_sort(Graph *g, Boolean top_down)`
Returns a list of nodes of graph in topologically sorted order.

Nodes

Type: `Node` **Include file:** `#include<HLS/graphlib.h>`

Description: A node in a `Graph` data-structure. The nodelist of a `Graph` is a list of these data-structures. Like graphs, node can have attributes, properties, and user information. Attributes have string values. Properties have values that can be any type of data-structure. One user information data-structure can be attached to each node. In addition, nodes have lists of incoming directed edges (`inedges`), outgoing directed edges (`outedges`) and incident undirected edges (`uedges`).

`Node *node_new(const char *nodename)`
Create new node.
`void node_free(Node *n)`
Frees node and constituent data structures.
`Graph *node_get_owner_graph(Node *n)`
Returns the graph that this node is a part of.
`void node_setattr(Node *n, const char *attr, const char *val)`
Sets attribute of node.
`char *node_getattr(Node *n, const char *attr)`
Gets the value of attribute of node.
`void node_delattr(Node *n, const char *attr`
Deletes a node attribute.
`void node_setprop(Node *n, const char *prop_name, void *data, void (*freeFunc) (void *)`
Sets value data structure of node property. If `freeFunc()` is defined, it will be applied to `data` when node_delprop() is called. `graph_free()` calls it too if node belongs to graph.
`void *node_getprop(Node *n, const char *prop_name)`
Returns pointer to value data structure of node property.
`void node_delprop(Node *n, const char *prop_name)`
Deletes a node property. Applies `freeFunc()` to value data structure if `freeFunc` is not `NULL`.
`void node_set_user_info(Node *n, void *info)`
Sets value data structure of node's user information element.
`void *node_get_user_info(Node *n)`
Returns pointer to value data structure that is user information.
`void node_clear_user_info(Node *n)`
Clears node's user information structure element.
`void node_addinedge(Node *n, Edge *e)`
Adds an incoming directed edge to node.
`void node_addoutedge(Node *n, Edge *e)`
Adds an outgoing directed edge to node.
`void node_adduedge(Node *n, UEdge *e)`
Adds an incident undirected edge to node.
`char *node_name(Node *n)`
Returns pointer to node name (private).
`List *node_inedges(Node *n)`
Returns pointer to list (private) of incoming directed edges.
`List *node_outedges(Node *n)`
Returns pointer to list (private) of outgoing directed edges.
`List *node_alledges(Node *n)`
Returns a newly-created list of all directed edges connected to node n.
`List *node_uedges(Node *n)`

Returns pointer to list (private) of incident undirected edges.
`List *node_parents(Node *n)`
Returns a pointer to a list of "parent" nodes of the node. These are nodes connected to it through an incoming edge. List must be freed by calling function.
`List *node_children(Node *n)`
Returns a pointer to a list of "child" nodes of the node. These are nodes connected to it through an outgoing edge. List must be freed by calling function.
`List *node_uneighbors(Node *n)`
Returns a pointer to a list of "neighbor" nodes of a node. These are nodes connected to it through an undirected edge. List must be freed by calling function.
`List *node_neighbors(Node *n)`
Returns a pointer to a list of all nodes connected to n by a directed edge. List must be freed by calling function.
`List *node_ancestors(Node *n)`
Returns a pointer to a list of "ancestor" nodes of the specified node. These are nodes that have paths through directed edges *to* the specified node. List must be freed by calling function.
`List *node_descendants(Node *n)`
Returns a pointer to a list of "ancestor" nodes of the specified node. These are nodes that have paths through directed edges *from* the specified node. List must be freed by calling function.
`Boolean node_is_parent(Node *n1, Node *n2)`
Returns `TRUE` if `n2` is a parent of `n1`. `FALSE` otherwise.
`Boolean node_is_child(Node *n1, Node *n2)`
Returns `TRUE` if `n2` is a child of `n1`. `FALSE` otherwise.
`Boolean node_is_ancestor(Node *n1, Node *n2)`
Returns `TRUE` if `n2` is a ancestor of `n1`. `FALSE` otherwise.
`Boolean node_is_descendant(Node *n1, Node *n2)`
Returns `TRUE` if `n2` is a descendant of `n1`. `FALSE` otherwise.
`void node_sforeach_attr(Node *n, char *attr_name, char *attr_val) { ... } node_endfor_attr;`
Looping construct. Iterates over all attributes of node. Attribute name (private) and value (private) are placed in `attr_name` and `attr_val`. Safe to delete attributes inside loop. Slow. `break`'ing ok. Do not `return` out of loop.
`void node_sforeach_prop(Node *n, char *prop_name, void *prop_val) { ... } node_endfor_prop;`
Looping construct. Iterates over all properties of node. Property name (private) and value are placed in `prop_name` and `prop_val`. Safe to delete properties inside loop. Slow. `break`'ing ok. Do not `return` out of loop.

Edges

Type: `Edge` **Include file:** #include<HLS/graphlib.h>

Description: A directed edge of a `Graph` data structure. The edge has attributes, properties, and a user information structure as do graphs and nodes. In addition, a directed edge has a source node and a destination node.

`Edge *edge_new(const char *name)`
Creates a new edge.
`void edge_free(Edge *e)`
Deletes edge and constituent data structures.
`Graph *edge_get_owner_graph(Edge *e)`
Returns the graph that this edge is a part of.
`void edge_setsrc(Edge *e, Node *n)`
Sets the source node of an edge.
`void *edge_delsrc(Edge *e)`
Clears the source node of an edge.
`void edge_setdst(Edge *e, Node *n)`
Sets the destination node of edge.
`void edge_deldst(Edge *e)`
Clears the destination node of edge.
`char *edge_name(Edge *e)`
Returns the edge name (private).
`Node *edge_src(Edge *e)`
Returns pointer to source node of edge.
`Node *edge_dst(Edge *e)`
Returns pointer to destination node of edge.
`void edge_setattr(Edge *e, const char *attr, const char *val)`
Sets the value of an attribute of edge.
`char *edge_getattr(Edge *e, const char *attr)`
Gets the string value of an attribute of edge.
`void edge_delattr(Edge *e, const char *attr)`
Deletes an edge attribute.
`void edge_setprop(Edge *e, const char *prop_name, void *data, void (*freeFunc) (void *))`
Sets the value of a property of an edge. The value is a pointer to a data-structure. If `freeFunc` is not `NULL` then it is applied to `data` when `edge_delprop()` is called. `graph_free()` applies it as well.
`void *edge_getprop(Edge *e, const char *prop_name)`
Returns the value of an edge property.
`void edge_delprop(Edge *e, const char *prop_name)`
Deletes an edge property. Applies `freeFunc`, if it is not `NULL`, to the data-structure that is the value.
`void edge_set_user_info(Edge *e, void *info)`
Sets the edge's user information data structure.
`d *edge_get_user_info(Edge *e)`
Returns a pointer to edge's user information data structure.
`void edge_clear_user_info(Edge *e)`
Clears the edge's user information data structure.
`void edge_sforeach_attr(Edge *e, char *attr_name, char *attr_val) { ... } edge_endfor_attr;`
Looping construct. Iterates over all attributes of edge. Attribute name (private) and value (private) are placed in `attr_name` and `attr_val`. Safe to delete attributes inside loop. Slow. `break`'ing ok. Do not `return` out of loop.
`void edge_sforeach_prop(Edge *, char *prop_name, void *prop_val) { ... } edge_endfor_prop;`
Looping construct. Iterates over all properties of edge. Property name (private) and value are placed in `prop_name` and `prop_val`. Safe to delete properties inside loop. Slow. `break`'ing ok. Do not `return` out of loop.

UEdges

Type: `UEdge` **Include file:** #include<HLS/graphlib.h>

Description: An undirected edge of a `Graph` data structure. The edge has attributes, properties, and a user information structure as do graphs and nodes. In addition, an undirected edge has two endpoint nodes.

`UEdge *uedge_new(const char *name)`
Creates a new uedge.
`void uedge_free(UEdge *ue)`
Deletes uedge and constituent data structures.
`Graph *uedge_get_owner_graph(UEdge *ue)`
Returns the graph that this uedge is a part of.
`void uedge_setn1(UEdge *ue, Node *n)`
Sets endpoint 1 of an uedge.
`void *uedge_deln1(UEdge *ue)`
Clears the endpoint 2 of an uedge.
`void uedge_setn2(UEdge *ue, Node *n)`

Sets the endpoint 2 of uedge.
`void uedge_deln2(UEdge *ue)`
Clears the endpoint 2 of uedge.
`char *uedge_name(UEdge *ue)`
Returns the uedge name (private).
`Node *uedge_src(UEdge *ue)`
Returns pointer to endpoint 2 of uedge.
`Node *uedge_dst(UEdge *ue)`
Returns pointer to endpoint 2 of uedge.
`void uedge_setattr(UEdge *ue, const char *attr, const char *val)`
Sets the value of an attribute of uedge.
`char *uedge_getattr(UEdge *ue, const char *attr)`
Gets the string value of an attribute of uedge.
`void uedge_delattr(UEdge *ue, const char *attr)`
Deletes an uedge attribute.
`void uedge_setprop(UEdge *ue, const char *prop_name, void *data, void (*freeFunc) (void *))`
Sets the value of a property of an uedge. The value is a pointer to a data-structure. If `freeFunc` is not `NULL` then it is applied to `data` when `uedge_delprop()` is called. `graph_free()` applies it as well.
`void *uedge_getprop(UEdge *ue, const char *prop_name)`
Returns the value of an uedge property.
`void uedge_delprop(UEdge *ue, const char *prop_name)`
Deletes an uedge property. Applies `freeFunc`, if it is not `NULL`, to the data-structure that is the value.
`void uedge_set_user_info(UEdge *ue, void *info)`
Sets the uedge's user information data structure.
`d *uedge_get_user_info(UEdge *ue)`
Returns a pointer to uedge's user information data structure.
`void uedge_clear_user_info(UEdge *ue)`
Clears the uedge's user information data structure.
`void uedge_sforeach_attr(UEdge *ue, char *attr_name, char *attr_val) { ... } uedge_endfor_attr;`
Looping construct. Iterates over all attributes of uedge. Attribute name (private) and value (private) are placed in `attr_name` and `attr_val`. Safe to delete attributes inside loop. Slow. `break`'ing ok. Do not `return` out of loop.
`void uedge_sforeach_prop(UEdge *ue, char *prop_name, void *prop_val) { ... } uedge_endfor_prop;`
Looping construct. Iterates over all properties of uedge. Property name (private) and value are placed in `prop_name` and `prop_val`. Safe to delete properties inside loop. Slow. `break`'ing ok. Do not `return` out of loop.

Tables

Type: `Table` **Include file:** `#include<HLS/generic.h>`

Description: Data structure for pretty-printing tables. A table has columns and rows. The user specifies the content string for each $(row, column)$ pair. The user can also specify column and row separator strings between columns. Each string can be multi-lined. The interface takes that into account. Columns are identified by integers in the range $[0, num\_columns - 1]$. Rows are identified by integers in the range $[0, num\_rows - 1]$.

`Table *table_new(unsigned int rows, unsigned int columns)`
Creates and initializes new table.
`void table_free(Table *t)`
Frees a table and constituent data structures.
`int table_rows(Table *t)`
Returns number of rows in table.
`int table_cols(Table *t)`
Returns number of columns in table.
`void table_set_entry(Table *t, int row, int col, const char *entry)`
Sets the (row, col) entry in the table.
`char *table_get_entry(Table *t, int row, int col)`
Gets the (row, col) entry in the table.
`void table_set_col_sep(Table *t, int col, const char *separator)`
Sets the column separator between columns col and $col + 1$. Default separator is `NULL` and denotes a single space between columns.
`char *table_get_col_sep(Table *t, int col)`
Gets the columns separator between columns col and $col + 1$.
`void table_set_row_sep(Table *t, int row, const char *separator)`
Sets the column separator between rows row and $row + 1$.
`char *table_get_row_sep(Table *t, int row)`
Gets the columns separator between rows row and $row + 1$.
`void table_set_entry_justification(Table *t, int row, int col, int justification)`
Sets the justification of the (row, col) entry. $0 =$ left-justified (default), $1 =$ centered, and $2 =$ right-justified.
`int table_get_entry_justification(Table *t, int row, int col)`
Gets the justification of the (row, col) entry.
`void table_fprintf(FILE *fp, Table *t)`
Pretty-prints the table to a stream.

Bit Matrices

Type: `Bit_Matrix` **Include file:** `#include<HLS/generic.h>`

Description: Maintains a matrix of bits. The size of the matrix is specified by the number of rows and the number of columns. Bits of the matrix are packed into integers to reduce storage space. Default value of bits is 0.

`Bit_Matrix *bit_matrix_new(int rows, int cols)`
Creates and initializes a new bit matrix.
`void bit_matrix_free(Bit_Matrix *bm)`
Frees a bit matrix and constituent data structures.
`int bit_matrix_rows(Bit_Matrix *bm)`
Returns the number of rows in bit matrix.
`int bit_matrix_cols(Bit_Matrix *bm)`
Returns the number of columns in bit matrix.
`void bit_matrix_set(Bit_Matrix *bm, int row, int col, int val)`
Set the value of the (row, col) bit. `val` can be 0 or 1.
`int bit_matrix_get(Bit_Matrix *bm, int row, int col)`
Get the value of the (row, col) bit.

Miscellaneous functions

Include file: `#include<HLS/generic.h>`

`pause_til_keypress()`
Pauses the program till the user hits the RETURN key.
`Boolean streq(const char *a, const char *b)`
Returns `TRUE` if both strings are same. `FALSE` otherwise.
`char *itoa(int d)`
Converts the integer to a string. String must be freed by caller.

Reservation Tables

Type: `RTable` **Include file:** `#include<HLS/rsrclib.h>`

Description: A resource reservation table. The table's entries are pointers to nodes. The nodes represent operations of the synthesized behavior. The columns represent resource instances (`RInstance`) on which operations can be executed. The rows represent control steps. Therefore, each table entry corresponds to the reservation status of a resource instance (column index) at a cstep (row index). A `NULL` entry means the resource instance is free at that cstep. Otherwise, the entry indicates the node using the resource instance at that time.

The `RTable` is built hierarchically. It contains a list of `RType` data structures defining the resource types available. Each `RType` contains a list of `RInstance` data structures representing individual instances of that resource type. Each `RInstance` contains an array for its column of the `RTable`.

`RTable *rtable_new(const char *name, int max_csteps)`
Creates and initializes a named rtable with specified csteps.

`void rtable_free(RTable *rtable)`
Frees up a resource table and constituents.

`void rtable_add_rtype(RTable *rtable, const char *rtype_name)`
Adds a resource type to an rtable.

`List *rtable_rtypes(RTable *rtable)`
Returns pointer to private list of resource types of rtable.

`Boolean rtable_has_rtype(RTable *rtable, const char *rtype_name)`
Returns `TRUE` if rtable has specified resource type. `FALSE` otherwise.

`void rtable_add_rinstances(RTable *rtable, const char *rtype_name, int number_of_instances)`
Adds specified number of instances of specified resource type to rtable. Instances are named by their type and instance number. E.g. add_0, mult_4, ...

`RInstance *rtable_add_rinstance(RTable *rtable, const char *rtype_name, const char *rinstance_name)`
Adds a named instance of specified resource type to the rtable.

`RInstance *rtable_get_rinstance(RTable *rtable, const char *rtype_name, const char *rinstance_name)`
Returns pointer to named instance of specified resource type in rtable. `NULL` if not found.

`int rtable_num_free_rinstances(RTable *rtable, const char *rtype_name, int cstep)`
Returns the number of unreserved resource instances of specified type at specified cstep. -1 if type is not in rtable.

`List *rtable_get_free_rinstances(RTable *rtable, const char *rtype_name, int cstep)`
Returns a list of unreserved instances (`RInstance *`) of specified type at specified cstep. `NULL` if type is not in rtable. List must be freed by caller.

`List *rtable_get_rinstances(RTable *rtable)`
Returns a list of all instance (`RInstance *`) of the rtable. List must be freed by caller.

`int rtable_num_all_rinstances(RTable *rtable, const char *rtype_name)`
Returns the number of instance of specified resource type in rtable. -1 if type is not in table.

`List *rtable_get_all_rinstances(RTable *rtable, const char *rtype_name)`
Returns a list of all instances(`RInstance *`) of specified resource type in rtable. `NULL` if type is not in table. List must be freed by caller.

`RType *rtable_get_rtype(RTable *rtable, const char *rtype_name)`
Returns pointer to private `RType` data structure of specified resource type. `NULL` if type not in table.

`int rtable_max_csteps(RTable *rtable)`
Returns the maximum number of csteps available on the rtable.

`void rtable_set_max_csteps(RTable *rtable, int max_csteps)`
Sets the maximum number of csteps of the rtable, truncating prior reservations if new number is smaller than old one.

Resource Types

Type: `RType` **Include file:** `#include<HLS/rsrclib.h>`

Description: This defines a type of resource available in a `RTable`. It contains a list of instances (`RInstance`) of resources of that type. `RType`s may be created independent of an `RTable` if needed.

`RType *rtype_new(const char *name, int max_csteps)`
Creates and initializes a resource type with specified csteps.

`void rtype_free(RType *rtype)`
Frees and resource type and constituents.

`char *rtype_name(RType *rtype)`
Returns the private name string of the rtype.

`void rtype_add_rinstances(RType *rtype, int num)`
Adds specified number of instances named by type and number, i.e. add_0, mult_4, ...

`RInstance *rtype_add_rinstance(RType *rtype, const char *rinstance_name)`
Adds a named instance to the rtype.

`int rtype_num_rinstances(RType *rtype)`
Returns the number of instances owned by the rtype.

`List *rtype_get_all_rinstances(RType *rtype)`
Returns a list of all instances owned by the rtype. List must be freed by caller.

`List *rtype_rinstances(RType *rtype)`
Returns pointer to private list of instances owned by rtype.

`RInstance *rtype_get_rinstance(RType *rtype, const char *rinstance_name)`
Returns pointer to named instance of rtype. `NULL` if rtype has no instance of specified name.

`int rtype_num_free_rinstances(RType *rtype, int cstep)`
Returns number of unreserved instances of rtype at specified cstep.

`List *rtype_get_free_rinstances(RType *rtype, int cstep)`
Returns a list of unreserved instances `(RInstance *) at specified cstep. List must be freed by caller.`

`int rtype_get_max_csteps(RType *rtype)`
Returns maximum number of csteps available to instances of rtype.

`void rtype_set_max_csteps(RType *rtype, int max_csteps)`
Sets the maximum number of csteps available to instances owned by rtype, truncating prior reservations if new number is smaller than old one.

Resource Instances

Type: `RInstance` **Include file:** `#include<HLS/rsrclib.h>`

Description: An instance of a resource of a specific type. Each resource instance is named and has a resource type name. Each also maintains an array of node pointers. The size of the array is the number of control steps in the design the resource instance is in. Each array element is a pointer to a (`Node`) of the dataflow graph (`Graph`) of the design. The index of the array element is the control step that array element represents. A `NULL` in the element means the resource instance is free at that cstep, otherwise it points to the `Node` using the resource instance at that time. Each resource instance also has a user-information element which is a pointer to a user-defined data structure.

`RInstance *rinstance_new(const char *name, const char *rtypename, int max_csteps)`
Creates and initializes a new resource instance of specified type and csteps.

`void rinstance_free(RInstance *rinstance)`
Frees up the memory associated with the resource instance.

`char *rinstance_name(RInstance *rinstance)`
Returns the private name of the resource instance.

`char *rinstance_rtype_name(RInstance *rinstance)`

Returns the private resource type of the resource instance.
`int rinstance_max_csteps(RInstance *rinstance)`
Returns the maximum number of csteps available.
`void rinstance_schedule(RInstance *rinstance, Node *n, int cstep)`
Schedules specified node to execute on resource instance at specified cstep.
`Node *rinstance_get_schedule(RInstance *rinstance, int cstep)`
Returns node scheduled at specified cstep on resource instance.
`void rinstance_unschedule(RInstance *rinstance, int cstep`
Frees up the resource instance for scheduling at specified cstep.
`Boolean rinstance_is_free(RInstance *rinstance, int cstep)`
Returns `TRUE` if resource instance is free at specified cstep. `FALSE` otherwise.
`void rinstance_set_max_csteps(RInstance *rinstance, int max_csteps)`
Adjusts the maximum number of csteps available to resource instance. Prior scheduling reservations are truncated as necessary.
`int rinstance_used_csteps(RInstance *rinstance)`
Returns the number of csteps this resource instance is used at.
`float rinstance_utilization(RInstance *rinstance)`
Returns the fraction of the used csteps to the maximum available.
`void rinstance_set_user_info(RInstance *r, void *i)`
Sets the user-defined data structure annotating resource instance.
`void *rinstance_get_user_info(RInstance *r)`
Returns the user information structure.
`void rinstance_clear_user_info(RInstance *r)`
Clears the user information field of resource instance.

Resource Allocation

Type: `RAllocation` **Include file:** `#include<HLS/rsrclib.h>`

Description: Defines a resource allocation which is a list of pairs: a resource type and the number of instances of that type allocated.

`RAllocation *rallocation_new(const char *rallocation_name)`
Creates and initializes a new resource allocation.
`void rallocation_free(RAllocation *rallocation`
Frees up a resource allocation and its constituents.
`void rallocation_set_alloc(RAllocation *rallocation, const char *rtype_name, int max_num_rinstances)`
Sets the allocation for specified resource type.
`int rallocation_get_alloc(RAllocation *rallocation, const char *rtype_name)`
Gets the allocation for specified resource type. -1 if type does not exist in rallocation.

Graphs (HLS-specific)

Include file: `#include<HLS/hlsinfolib.h>`

Description: Each dataflow graph (DFG) is represented by a `Graph`. Each `Graph` has an HLS-specific information data structure as an element. This element of a `Graph` is initialized only when the HLS-specific interface is used. It includes a resource allocation, a resource reservation table, the maximum number of csteps available, and the period of a cstep in technology library time units.

`Boolean graph_set_rallocation(Graph *g, RAllocation *rallocation)`
Sets the resource allocation for a graph. If a node of the graph has a resource type not in the rallocation, the user is prompted for an allocation. If `rallocation` is `NULL`, then the function creates a resource allocation and queries the user for input. If it is not `NULL`, it becomes part of the graph and is owned by it.
`void graph_free_rallocation(Graph *g)`
Frees the graph's resource allocation data structure.
`RAllocation *graph_get_rallocation(Graph *g)`
Returns pointer to the private resource allocation of the graph.
`int graph_get_rtype_alloc (Graph *g, const char *rtype_name)`
Returns the allocation of specified resource type in graph.
`Boolean graph_init_rtable(Graph *g, int max_csteps)`
Initializes the resource table of a graph and sets the maximum number of csteps available to the graph. Destroys previously existing rtable, if one exists.
`void graph_free_rtable(Graph *g)`
Frees up the resource reservation table of the graph.
`RTable *graph_rtable(Graph *g)`
Returns pointer to the private resource reservation table of the graph.
`void graph_unschedule(Graph *g)`
Unschedules all nodes of the graph.
`Boolean graph_set_max_csteps(Graph *g, int max_csteps)`
Sets the maximum number of csteps available to the graph. Truncates previous schedules if new number is smaller than old one.
`int graph_get_max_csteps(Graph *g)`
Returns the maximum number of csteps available to the graph.
`void graph_print_schedule(Graph *g)`
Pretty-prints the resource reservation table of a graph.
`void graph_register_hls_info(Graph *g)`
Registers HLS-specific information of a graph as attributes. This enables the information to be saved on a call to the `graph_write()` functions. The attributes used are "max_csteps" for the graph and "cstep", "rinstance", "operation", and "resource" for each node to specify scheduling, binding, operation-type, and resource-type information.
`void graph_retrieve_hls_info(Graph *g)`
Recovers the HLS-specific information in a graph as saved by `graph_register_hls_info()`.
`Boolean graph_is_scheduled(Graph *g)`
Returns `TRUE` if all nodes of the graph are scheduled. `FALSE` otherwise.
`Boolean graph_is_bound(Graph *g)`
Returns `TRUE` if all nodes of the graph are bound. `FALSE` otherwise.

Nodes (HLS-specific)

Include file: `#include<HLS/hlsinfolib.h>`

Description: Each node of a graph has an HLS-specific information data structure which is initialized only when the HLS-specific interface of libHLS is used. This information includes the cstep the node is scheduled at, the resource instance it is bound to, the operation type of the node and the resource type of the node.

`Boolean node_schedule (Node *n, int cstep)`
Schedules the node at specified cstep. Returns `TRUE` if it succeeds and `FALSE` if it fails. Failures happens when the node is already bound and the resource instance is reserved at the specified cstep. It also fails if the node is already scheduled at a different cstep.
`void node_unschedule(Node *n)`
Unschedules node.
`Boolean node_reschedule(Node *n, int cstep)`
Re-schedules a previously scheduled node. Same return conditions as `node_schedule()`.
`int node_get_cstep(Node *n)`
Returns the cstep the node is scheduled at. -1 if node is not scheduled.
`Boolean node_is_scheduled(Node *n)`
Returns `TRUE` if the node is scheduled and `FALSE` if it not.
`Boolean node_bind(Node *n, RInstance *rinstance)`
Binds the node to the specified resource instance. Returns `TRUE` if it succeeds. Returns `FALSE` on failures which happen if the node is already bound to a different resource instance or if the node is scheduled and the specified resource instance is busy at that cstep.
`void node_unbind(Node *n)`
Unbinds node.

`Boolean node_rebind(Node *n, RInstance *rinstance)`
Rebinds previously bound node. Return conditions same as in `node_bind()`.

`RInstance *node_get_rinstance(Node *n)`
Returns the resource instance the node is bound to. `NULL` if the node is not bound.

`Boolean node_is_bound(Node *n)`
Returns `TRUE` if the node is bound and `FALSE` if it isn't.

`char *node_get_rtype_name(Node *n)`
Returns the private resource type name for the node. `NULL` if the node requires no resource.

`void node_set_rtype_name(Node *n, char *rtype_name)`
Sets the resource type name for the node.

`void node_clear_rtype_name(Node *n)`
Clears the resource type name for the node.

`char *node_get_op_name(Node *n)`
Returns the private operation type name of the node. `NULL` if the node has no operation.

`void node_set_op_name(Node *n, char *op_name)`
Sets the operation type name of the node.

`void node_clear_op_name(Node *n)`
Clears the operation type name of the node.

`int node_get_asap_cstep(Node *n)`
Returns the ASAP cstep of the timeframe of node. -1 if value is invalid i.e. node is now unscheduleable.

`int node_get_alap_cstep(Node *n)`
Returns the ALAP cstep of the timeframe of node. -1 if value is invalid i.e. node is now unscheduleable.