# Contents

1 Preface ................................................. 1
   1.1 Scope of the Document ............................. 1
   1.2 Audience .......................................... 1
   1.3 Document Status .................................. 1
   1.4 License ............................................ 1

2 Introduction .......................................... 3
   2.1 Motivation ......................................... 3
   2.2 UCX ................................................. 3

3 Design ................................................... 5
   3.1 UCS .................................................. 5
   3.2 UCT .................................................. 5
   3.3 UCP .................................................. 6

4 Conventions and Notations ................................ 7
   4.1 Blocking Behavior .................................. 7
   4.2 Non-blocking Behavior ............................ 7
   4.3 Fairness ............................................ 7
   4.4 Interaction with Signal Handler Functions ...... 7

5 Deprecated List ........................................ 9

6 Module Documentation .................................. 11
   6.1 Unified Communication Protocol (UCP) API .... 11
      6.1.1 Detailed Description ......................... 11
   6.2 UCP Application Context .......................... 12
      6.2.1 Detailed Description ......................... 13
      6.2.2 Data Structure Documentation .............. 13
         6.2.2.1 struct ucp_context_attr ................ 13
         6.2.2.2 struct ucp_tag_recv_info ............... 13
         6.2.2.3 struct ucp_request_param_t .......... 14
         6.2.2.4 union ucp_request_param_t.cb .......... 15
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2.2.5</td>
<td>15</td>
</tr>
<tr>
<td>6.2.3</td>
<td>15</td>
</tr>
<tr>
<td>6.2.3.1</td>
<td>15</td>
</tr>
<tr>
<td>6.2.3.2</td>
<td>15</td>
</tr>
<tr>
<td>6.2.3.3</td>
<td>15</td>
</tr>
<tr>
<td>6.2.3.4</td>
<td>16</td>
</tr>
<tr>
<td>6.2.3.5</td>
<td>16</td>
</tr>
<tr>
<td>6.2.4</td>
<td>16</td>
</tr>
<tr>
<td>6.2.4.1</td>
<td>16</td>
</tr>
<tr>
<td>6.2.4.2</td>
<td>16</td>
</tr>
<tr>
<td>6.2.4.3</td>
<td>17</td>
</tr>
<tr>
<td>6.2.5</td>
<td>17</td>
</tr>
<tr>
<td>6.2.5.1</td>
<td>17</td>
</tr>
<tr>
<td>6.2.5.2</td>
<td>17</td>
</tr>
<tr>
<td>6.2.5.3</td>
<td>18</td>
</tr>
<tr>
<td>6.2.5.4</td>
<td>18</td>
</tr>
<tr>
<td>6.2.5.5</td>
<td>19</td>
</tr>
<tr>
<td>6.2.5.6</td>
<td>19</td>
</tr>
<tr>
<td>6.3</td>
<td>20</td>
</tr>
<tr>
<td>6.3.1</td>
<td>22</td>
</tr>
<tr>
<td>6.3.2</td>
<td>22</td>
</tr>
<tr>
<td>6.3.2.1</td>
<td>22</td>
</tr>
<tr>
<td>6.3.2.2</td>
<td>23</td>
</tr>
<tr>
<td>6.3.2.3</td>
<td>24</td>
</tr>
<tr>
<td>6.3.2.4</td>
<td>24</td>
</tr>
<tr>
<td>6.3.2.5</td>
<td>24</td>
</tr>
<tr>
<td>6.3.2.6</td>
<td>25</td>
</tr>
<tr>
<td>6.3.2.7</td>
<td>25</td>
</tr>
<tr>
<td>6.3.2.8</td>
<td>25</td>
</tr>
<tr>
<td>6.3.2.9</td>
<td>26</td>
</tr>
<tr>
<td>6.3.3</td>
<td>26</td>
</tr>
<tr>
<td>6.3.3.1</td>
<td>26</td>
</tr>
<tr>
<td>6.3.3.2</td>
<td>26</td>
</tr>
<tr>
<td>6.3.3.3</td>
<td>26</td>
</tr>
<tr>
<td>6.3.3.4</td>
<td>26</td>
</tr>
<tr>
<td>6.3.3.5</td>
<td>27</td>
</tr>
<tr>
<td>6.3.3.6</td>
<td>27</td>
</tr>
<tr>
<td>6.3.3.7</td>
<td>27</td>
</tr>
<tr>
<td>6.3.3.8</td>
<td>27</td>
</tr>
<tr>
<td>6.3.3.9</td>
<td>27</td>
</tr>
</tbody>
</table>
6.6.3 Typedef Documentation ........................................ 58
  6.6.3.1 ucp_stream_poll_ep_t .................................. 58
  6.6.3.2 ucp_ep_h ............................................. 58
  6.6.3.3 ucp_conn_request_h .................................. 58
  6.6.3.4 ucp_am_callback_t .................................. 59
  6.6.3.5 ucp_am_recv_callback_t ................................. 59
  6.6.3.6 ucp_ep_params_t .................................... 60

6.6.4 Enumeration Type Documentation .......................... 60
  6.6.4.1 ucp_ep_params_field ................................ 60
  6.6.4.2 ucp_ep_params_flags_field ........................... 60
  6.6.4.3 ucp_ep_close_flags_t ................................ 61
  6.6.4.4 ucp_ep_close_mode .................................. 61
  6.6.4.5 ucp_cb_param_flags .................................. 62
  6.6.4.6 ucp_err_handling_mode_t .............................. 62

6.6.5 Function Documentation .................................... 62
  6.6.5.1 ucp_ep_create() .................................... 62
  6.6.5.2 ucp_ep_close_nb() ................................... 63
  6.6.5.3 ucp_ep_close_nbx() .................................. 63
  6.6.5.4 ucp_ep_print_info() .................................. 64
  6.6.5.5 ucp_ep_flush_nb() .................................... 64
  6.6.5.6 ucp_ep_flush_nbx() ................................... 65
  6.6.5.7 ucp_request_release() ................................ 65
  6.6.5.8 ucp_ep_destroy() ..................................... 66
  6.6.5.9 ucp_disconnect_nb() .................................. 66
  6.6.5.10 ucp_request_test() .................................. 66
  6.6.5.11 ucp_ep_flush() ...................................... 66
  6.6.5.12 ucp_ep_modify_nb() .................................. 67

6.7 UCP Communication routines ................................. 68
  6.7.1 Detailed Description .................................... 71
  6.7.2 Data Structure Documentation ........................... 72
    6.7.2.1 struct ucp_err_handler .............................. 72
  6.7.3 Typedef Documentation ................................ 72
    6.7.3.1 ucp_tag_t ......................................... 72
    6.7.3.2 ucp_tag_message_h ................................ 72
    6.7.3.3 ucp_datatype_t ..................................... 72
    6.7.3.4 ucp_send_callback_t ................................ 72
    6.7.3.5 ucp_send_nbx_callback_t ............................. 73
    6.7.3.6 ucp_err_handler_cb_t ................................ 73
    6.7.3.7 ucp_err_handler_t ................................... 73
    6.7.3.8 ucp_stream_recv_callback_t ......................... 73
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.7.3.9</td>
<td>ucp_stream_recv_nbx_callback_t</td>
</tr>
<tr>
<td>6.7.3.10</td>
<td>ucp_tag_recv_callback_t</td>
</tr>
<tr>
<td>6.7.3.11</td>
<td>ucp_tag_recv_nbx_callback_t</td>
</tr>
<tr>
<td>6.7.3.12</td>
<td>ucp_am_recv_data_nbx_callback_t</td>
</tr>
<tr>
<td>6.7.4</td>
<td>Enumeration Type Documentation</td>
</tr>
<tr>
<td>6.7.4.1</td>
<td>ucp_atomic_post_op_t</td>
</tr>
<tr>
<td>6.7.4.2</td>
<td>ucp_atomic_fetch_op_t</td>
</tr>
<tr>
<td>6.7.4.3</td>
<td>ucp_atomic_op_t</td>
</tr>
<tr>
<td>6.7.4.4</td>
<td>ucp_stream_recv_flags_t</td>
</tr>
<tr>
<td>6.7.4.5</td>
<td>ucp_op_attr_t</td>
</tr>
<tr>
<td>6.7.4.6</td>
<td>ucp_am_recv_attr_t</td>
</tr>
<tr>
<td>6.7.4.7</td>
<td>ucp_am_handler_param_field</td>
</tr>
<tr>
<td>6.7.5</td>
<td>Function Documentation</td>
</tr>
<tr>
<td>6.7.5.1</td>
<td>ucp_am_send_nb()</td>
</tr>
<tr>
<td>6.7.5.2</td>
<td>ucp_am_send_nbx()</td>
</tr>
<tr>
<td>6.7.5.3</td>
<td>ucp_am_recv_data_nbx()</td>
</tr>
<tr>
<td>6.7.5.4</td>
<td>ucp_am_data_release()</td>
</tr>
<tr>
<td>6.7.5.5</td>
<td>ucp_stream_send_nb()</td>
</tr>
<tr>
<td>6.7.5.6</td>
<td>ucp_stream_send_nbx()</td>
</tr>
<tr>
<td>6.7.5.7</td>
<td>ucp_tag_send_nb()</td>
</tr>
<tr>
<td>6.7.5.8</td>
<td>ucp_tag_send_nbr()</td>
</tr>
<tr>
<td>6.7.5.9</td>
<td>ucp_tag_send_sync_nb()</td>
</tr>
<tr>
<td>6.7.5.10</td>
<td>ucp_tag_send_nbx()</td>
</tr>
<tr>
<td>6.7.5.11</td>
<td>ucp_tag_send_sync_nbx()</td>
</tr>
<tr>
<td>6.7.5.12</td>
<td>ucp_stream_recv_nb()</td>
</tr>
<tr>
<td>6.7.5.13</td>
<td>ucp_stream_recv_nbx()</td>
</tr>
<tr>
<td>6.7.5.14</td>
<td>ucp_stream_recv_data_nb()</td>
</tr>
<tr>
<td>6.7.5.15</td>
<td>ucp_tag_recv_nb()</td>
</tr>
<tr>
<td>6.7.5.16</td>
<td>ucp_tag_recv_nbr()</td>
</tr>
<tr>
<td>6.7.5.17</td>
<td>ucp_tag_recv_nbx()</td>
</tr>
<tr>
<td>6.7.5.18</td>
<td>ucp_tag_probe_nb()</td>
</tr>
<tr>
<td>6.7.5.19</td>
<td>ucp_tag_msg_recv_nb()</td>
</tr>
<tr>
<td>6.7.5.20</td>
<td>ucp_tag_msg_recv_nbx()</td>
</tr>
<tr>
<td>6.7.5.21</td>
<td>ucp_put_nb()</td>
</tr>
<tr>
<td>6.7.5.22</td>
<td>ucp_put_nbx()</td>
</tr>
<tr>
<td>6.7.5.23</td>
<td>ucp_get_nb()</td>
</tr>
<tr>
<td>6.7.5.24</td>
<td>ucp_get_nbx()</td>
</tr>
<tr>
<td>6.7.5.25</td>
<td>ucp_get_nbr()</td>
</tr>
<tr>
<td>6.7.5.26</td>
<td>ucp_get_nbx()</td>
</tr>
<tr>
<td>6.7.5.27</td>
<td>ucp_atomic_post()</td>
</tr>
</tbody>
</table>
CONTENTS

6.7.5.28 ucp_atomic_fetch_nb() .................................................. 97
6.7.5.29 ucp_atomic_op_nbx() .................................................. 98
6.7.5.30 ucp_request_check_status() ........................................ 99
6.7.5.31 ucp_tag_recv_request_test() ....................................... 100
6.7.5.32 ucp_stream_recv_request_test() .................................... 100
6.7.5.33 ucp_request_cancel() .................................................. 101
6.7.5.34 ucp_stream_data_release() .......................................... 101
6.7.5.35 ucp_request_free() .................................................... 101
6.7.5.36 ucp_request_alloc() ................................................... 101
6.7.5.37 ucp_request_is_completed() ....................................... 102
6.7.5.38 ucp_put() ............................................................... 102
6.7.5.39 ucp_get() ............................................................... 103
6.7.5.40 ucp_atomic_add32() .................................................. 103
6.7.5.41 ucp_atomic_add64() .................................................. 104
6.7.5.42 ucp_atomic_fadd32() .................................................. 105
6.7.5.43 ucp_atomic_fadd64() .................................................. 106
6.7.5.44 ucp_atomic_swap32() .................................................. 106
6.7.5.45 ucp_atomic_swap64() .................................................. 107
6.7.5.46 ucp_atomic_cswap32() .............................................. 108
6.7.5.47 ucp_atomic_cswap64() .............................................. 109

6.8 UCP Configuration ............................................................... 110
  6.8.1 Detailed Description ..................................................... 110
  6.8.2 Data Structure Documentation ........................................ 110
  6.8.2.1 struct ucp_params .................................................. 110
  6.8.3 Typedef Documentation ................................................ 112
  6.8.3.1 ucp_params_t .......................................................... 112
  6.8.3.2 ucp_config_t .......................................................... 112
  6.8.4 Function Documentation ............................................... 112
  6.8.4.1 ucp_config_read() .................................................. 113
  6.8.4.2 ucp_config_release() .............................................. 113
  6.8.4.3 ucp_config_modify() .............................................. 113
  6.8.4.4 ucp_config_print() .................................................. 114

6.9 UCP Data type routines ...................................................... 115
  6.9.1 Detailed Description ..................................................... 116
  6.9.2 Data Structure Documentation ........................................ 116
  6.9.2.1 struct ucp_dt_iov ................................................... 116
  6.9.3 Macro Definition Documentation .................................... 116
  6.9.3.1 ucp_dt_make_contig ................................................. 116
  6.9.3.2 ucp_dt_make_iov .................................................... 117
  6.9.4 Typedef Documentation ............................................... 117

© 2021 Unified Communication X (UCX). All rights reserved.
6.9.4.1 ucp_dt_iov_t .................................................. 117
6.9.4.2 ucp_generic_dt_ops_t ........................................ 117
6.9.5 Enumeration Type Documentation .................................... 117
6.9.5.1 ucp_dt_type .......................................................... 117
6.9.6 Function Documentation .............................................. 118
6.9.6.1 ucp_dt_create_generic() .......................................... 118
6.9.6.2 ucp_dt_destroy() .................................................. 118
6.9.7 Variable Documentation ............................................... 119
6.9.7.1 start_pack .......................................................... 119
6.9.7.2 start_unpack ....................................................... 119
6.9.7.3 packed_size ....................................................... 119
6.9.7.4 pack ................................................................. 120
6.9.7.5 unpack ............................................................. 120
6.9.7.6 finish ............................................................... 120
6.10 Unified Communication Transport (UCT) API ............................ 121
6.10.1 Detailed Description .................................................. 121
6.11 UCT Communication Resource ............................................ 122
6.11.1 Detailed Description .................................................. 126
6.11.2 Data Structure Documentation ....................................... 126
6.11.2.1 struct uct_md_resource_desc ...................................... 126
6.11.2.2 struct uct_component_attr ........................................ 126
6.11.2.3 struct uct_tl_resource_desc ...................................... 127
6.11.2.4 struct uct_iface_attr .............................................. 128
6.11.2.5 struct uct_iface_attr.cap ......................................... 128
6.11.2.6 struct uct_iface_attr.cap.put ..................................... 128
6.11.2.7 struct uct_iface_attr.cap.get ..................................... 129
6.11.2.8 struct uct_iface_attr.cap.am ..................................... 129
6.11.2.9 struct uct_iface_attr.cap.tag ..................................... 129
6.11.2.10 struct uct_iface_attr.cap.tag.recv ............................... 129
6.11.2.11 struct uct_iface_attr.cap.tag.eager ............................. 130
6.11.2.12 struct uct_iface_attr.cap.tag.rndv .............................. 130
6.11.2.13 struct uct_iface_attr.cap.atomic32 ............................ 130
6.11.2.14 struct uct_iface_attr.cap.atomic64 ............................ 130
6.11.2.15 struct uct_iface_params .......................................... 130
6.11.2.16 union uct_iface_params.mode .................................... 131
6.11.2.17 struct uct_iface_params.mode.device ............................ 131
6.11.2.18 struct uct_iface_params.mode.sockaddr ........................ 132
6.11.2.19 struct uct_ep_params ............................................. 132
6.11.2.20 struct uct_completion ............................................ 133
6.11.2.21 struct uct_pending_req .......................................... 134

© 2021 Unified Communication X (UCX). All rights reserved.
6.11.2.22 struct uct_iov ............................................. 134
6.11.3 Typedef Documentation ........................................ 135
  6.11.3.1 uct_md_resource_desc_t .................................. 135
  6.11.3.2 uct_component_attr_t .................................... 135
  6.11.3.3 uct_tl_resource_desc_t .................................. 135
  6.11.3.4 uct_component_h .......................................... 135
  6.11.3.5 uct_iface_h .............................................. 135
  6.11.3.6 uct_iface_config_t ....................................... 135
  6.11.3.7 uct_md_config_t ......................................... 136
  6.11.3.8 uct_cm_config_t ......................................... 136
  6.11.3.9 uct_ep_h .................................................. 136
  6.11.3.10 uct_mem_h ............................................... 136
  6.11.3.11 uct_rkey_t .............................................. 136
  6.11.3.12 uct_md_h ................................................ 136
  6.11.3.13 uct_md_ops_t .......................................... 136
  6.11.3.14 uct_rkey_ctx_h ....................................... 136
  6.11.3.15 uct_iface_attr_t ....................................... 136
  6.11.3.16 uct_iface_params_t .................................... 137
  6.11.3.17 uct_md_attr_t ......................................... 137
  6.11.3.18 uct_completion_t ...................................... 137
  6.11.3.19 uct_pending_req_t .................................... 137
  6.11.3.20 uct_worker_h .......................................... 137
  6.11.3.21 uct_md_t ............................................... 137
  6.11.3.22 uct_am_trace_type_t .................................. 137
  6.11.3.23 uct_device_addr_t .................................... 137
  6.11.3.24 uct_iface_addr_t ...................................... 137
  6.11.3.25 uct_ep_addr_t ......................................... 138
  6.11.3.26 uct_ep_params_t ...................................... 138
  6.11.3.27 uct_cm_attr_t ......................................... 138
  6.11.3.28 uct_cm_t ............................................... 138
  6.11.3.29 uct_cm_h .............................................. 138
  6.11.3.30 uct_listener_attr_t .................................. 138
  6.11.3.31 uct_listener_h ....................................... 138
  6.11.3.32 uct_listener_params_t ................................. 138
  6.11.3.33 uct_tag_context_t .................................... 138
  6.11.3.34 uct_tag_t .............................................. 139
  6.11.3.35 uct_worker_cb_id_t .................................. 139
  6.11.3.36 uct_conn_request_h .................................. 139
  6.11.3.37 uct_iov_t .............................................. 139
  6.11.3.38 uct_completion_callback_t ............................. 139
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.11.3.39</td>
<td>uct_pending_callback_t</td>
</tr>
<tr>
<td>6.11.3.40</td>
<td>uct_error_handler_t</td>
</tr>
<tr>
<td>6.11.3.41</td>
<td>uct_pending_purge_callback_t</td>
</tr>
<tr>
<td>6.11.3.42</td>
<td>uct_pack_callback_t</td>
</tr>
<tr>
<td>6.11.3.43</td>
<td>uct_unpack_callback_t</td>
</tr>
<tr>
<td>6.11.3.44</td>
<td>uct_async_event_cb_t</td>
</tr>
<tr>
<td>6.11.4</td>
<td>Enumeration Type Documentation</td>
</tr>
<tr>
<td>6.11.4.1</td>
<td>uct_component_attr_field</td>
</tr>
<tr>
<td>6.11.4.2</td>
<td>anonymous enum</td>
</tr>
<tr>
<td>6.11.4.3</td>
<td>uct_device_type_t</td>
</tr>
<tr>
<td>6.11.4.4</td>
<td>uct_iface_event_types</td>
</tr>
<tr>
<td>6.11.4.5</td>
<td>uct_flush_flags</td>
</tr>
<tr>
<td>6.11.4.6</td>
<td>uct_progress_types</td>
</tr>
<tr>
<td>6.11.4.7</td>
<td>uct_cb_flags</td>
</tr>
<tr>
<td>6.11.4.8</td>
<td>uct_iface_open_mode</td>
</tr>
<tr>
<td>6.11.4.9</td>
<td>uct_iface_params_field</td>
</tr>
<tr>
<td>6.11.4.10</td>
<td>uct_ep_params_field</td>
</tr>
<tr>
<td>6.11.4.11</td>
<td>anonymous enum</td>
</tr>
<tr>
<td>6.11.4.12</td>
<td>uct_cb_param_flags</td>
</tr>
<tr>
<td>6.11.5</td>
<td>Function Documentation</td>
</tr>
<tr>
<td>6.11.5.1</td>
<td>uct_query_components()</td>
</tr>
<tr>
<td>6.11.5.2</td>
<td>uct_release_component_list()</td>
</tr>
<tr>
<td>6.11.5.3</td>
<td>uct_component_query()</td>
</tr>
<tr>
<td>6.11.5.4</td>
<td>uct_md_open()</td>
</tr>
<tr>
<td>6.11.5.5</td>
<td>uct_md_close()</td>
</tr>
<tr>
<td>6.11.5.6</td>
<td>uct_md_query_tl_resources()</td>
</tr>
<tr>
<td>6.11.5.7</td>
<td>uct_release_tl_resource_list()</td>
</tr>
<tr>
<td>6.11.5.8</td>
<td>uct_md_iface_config_read()</td>
</tr>
<tr>
<td>6.11.5.9</td>
<td>uct_config_release()</td>
</tr>
<tr>
<td>6.11.5.10</td>
<td>uct_iface_open()</td>
</tr>
<tr>
<td>6.11.5.11</td>
<td>uct_iface_close()</td>
</tr>
<tr>
<td>6.11.5.12</td>
<td>uct_iface_query()</td>
</tr>
<tr>
<td>6.11.5.13</td>
<td>uct_iface_get_device_address()</td>
</tr>
<tr>
<td>6.11.5.14</td>
<td>uct_iface_get_address()</td>
</tr>
<tr>
<td>6.11.5.15</td>
<td>uct_iface_is_reachable()</td>
</tr>
<tr>
<td>6.11.5.16</td>
<td>uct_ep_check()</td>
</tr>
<tr>
<td>6.11.5.17</td>
<td>uct_iface_event_fd_get()</td>
</tr>
<tr>
<td>6.11.5.18</td>
<td>uct_iface_event_arm()</td>
</tr>
<tr>
<td>6.11.5.19</td>
<td>uct_iface_mem_alloc()</td>
</tr>
<tr>
<td>6.11.5.20</td>
<td>uct_iface_mem_free()</td>
</tr>
</tbody>
</table>

© 2021 Unified Communication X (UCX). All rights reserved.
6.11.5.21 uct_ep_create() .................................................. 154
6.11.5.22 uct_ep_destroy() ................................................. 154
6.11.5.23 uct_ep_get_address() ........................................... 155
6.11.5.24 uct_ep_connect_to_ep() ....................................... 155
6.11.5.25 uct_iface_flush() ................................................ 155
6.11.5.26 uct_iface_fence() ............................................... 156
6.11.5.27 uct_ep_pending_add() ......................................... 156
6.11.5.28 uct_ep_pending_purge() ....................................... 157
6.11.5.29 uct_ep_flush() .................................................. 157
6.11.5.30 uct_ep_fence() ................................................. 157
6.11.5.31 uct_iface_progress_enable() ................................. 158
6.11.5.32 uct_iface_progress_disable() ............................... 158
6.11.5.33 uct_iface_progress() ......................................... 159
6.11.5.34 uct_completion_update_status() ............................ 159

6.12 UCT Communication Context ........................................ 160
6.12.1 Detailed Description ............................................. 160
6.12.2 Enumeration Type Documentation ............................... 160
6.12.2.1 uct_alloc_method_t ............................................. 160
6.12.3 Function Documentation ......................................... 161
6.12.3.1 uct_worker_create() .......................................... 161
6.12.3.2 uct_worker_destroy() ......................................... 161
6.12.3.3 uct_worker_progress_register_safe() ....................... 161
6.12.3.4 uct_worker_progress_unregister_safe() .................... 162
6.12.3.5 uct_config_get() ............................................... 162
6.12.3.6 uct_config_modify() ......................................... 163
6.12.3.7 uct_worker_progress() ....................................... 163

6.13 UCT Memory Domain ................................................. 165
6.13.1 Detailed Description ............................................. 166
6.13.2 Data Structure Documentation .................................. 167
6.13.2.1 struct uct_md_attr ........................................... 167
6.13.2.2 struct uct_md_attr.cap ..................................... 167
6.13.2.3 struct uct_md_mem_attr ...................................... 167
6.13.2.4 struct uct_allocated_memory ................................. 168
6.13.2.5 struct uct_rkey_bundle ...................................... 168
6.13.2.6 struct uct_mem_alloc_params_t ............................. 168
6.13.2.7 struct uct_mem_alloc_params_t.mds ........................ 169
6.13.3 Typedef Documentation .......................................... 169
6.13.3.1 uct_md_mem_attr_t .......................................... 169
6.13.3.2 uct_allocated_memory_t ..................................... 169
6.13.3.3 uct_rkey_bundle_t ........................................... 169
6.13.4 Enumeration Type Documentation .................................................. 169
  6.13.4.1 uct_sockaddr_accessibility_t .............................................. 169
  6.13.4.2 anonymous enum .............................................................. 170
  6.13.4.3 uct_md_mem_flags ............................................................. 170
  6.13.4.4 uct_mem_advice_t ................................................................ 171
  6.13.4.5 uct_md_mem_attr_field ....................................................... 171
  6.13.4.6 uct_mem_alloc_params_field_t ............................................. 171

6.13.5 Function Documentation ............................................................... 171
  6.13.5.1 uct_md_mem_query() ............................................................. 171
  6.13.5.2 uct_md_query() ................................................................ 172
  6.13.5.3 uct_md_mem_advise() ........................................................... 172
  6.13.5.4 uct_md_mem_reg() ............................................................... 173
  6.13.5.5 uct_md_mem_dereg() ............................................................. 173
  6.13.5.6 uct_md_detect_memory_type() ................................................ 174
  6.13.5.7 uct_mem_alloc() .................................................................. 174
  6.13.5.8 uct_mem_free() ................................................................... 174
  6.13.5.9 uct_md_config_read() ........................................................... 175
  6.13.5.10 uct_md_is_sockaddr_accessible() ......................................... 175
  6.13.5.11 uct_md_mkey_pack() ........................................................... 176
  6.13.5.12 uct_rkey_unpack() ............................................................. 176
  6.13.5.13 uct_rkey_ptr() .................................................................. 176
  6.13.5.14 uct_rkey_release() ............................................................. 177

6.14 UCT Active messages ........................................................................ 178
  6.14.1 Detailed Description .................................................................. 178
  6.14.2 Typedef Documentation ............................................................. 178
    6.14.2.1 uct_am_callback_t ......................................................... 178
    6.14.2.2 uct_am_tracer_t ............................................................. 179
  6.14.3 Enumeration Type Documentation .............................................. 179
    6.14.3.1 uct_msg_flags ................................................................. 179
    6.14.3.2 uct_am_trace_type ........................................................ 180
  6.14.4 Function Documentation ............................................................ 180
    6.14.4.1 uct_iface_set_am_handler() ........................................... 180
    6.14.4.2 uct_iface_set_am_tracer() .............................................. 180
    6.14.4.3 uct_iface_release_desc() ................................................ 181
    6.14.4.4 uct_ep_am_short() .......................................................... 181
    6.14.4.5 uct_ep_am_bcopy() .......................................................... 181
    6.14.4.6 uct_ep_am_zcopy() .......................................................... 182

6.15 UCT Remote memory access operations ........................................... 183
  6.15.1 Detailed Description .................................................................. 183
  6.15.2 Function Documentation ............................................................ 183

© 2021 Unified Communication X (UCX). All rights reserved.
6.19.2.4 UCT_IFACE_FLAG_PENDING ........................................ 213
6.19.2.5 UCT_IFACE_FLAG_PUT_SHORT ........................................ 213
6.19.2.6 UCT_IFACE_FLAG_PUT_BCOPY ........................................ 213
6.19.2.7 UCT_IFACE_FLAG_PUT_ZCOPY ........................................ 213
6.19.2.8 UCT_IFACE_FLAG_GET_SHORT ........................................ 214
6.19.2.9 UCT_IFACE_FLAG_GET_BCOPY ........................................ 214
6.19.2.10 UCT_IFACE_FLAG_GET_ZCOPY ....................................... 214
6.19.2.11 UCT_IFACE_FLAG_ATOMIC_CPU ..................................... 214
6.19.2.12 UCT_IFACE_FLAG_ATOMIC_DEVICE .................................. 214
6.19.2.13 UCT_IFACE_FLAG_ERRHANDLE_SHORT_BUF ........................... 214
6.19.2.14 UCT_IFACE_FLAG_ERRHANDLE_BCOPY_BUF .......................... 214
6.19.2.15 UCT_IFACE_FLAG_ERRHANDLE_ZCOPY_BUF .......................... 214
6.19.2.16 UCT_IFACE_FLAG_ERRHANDLE_AM_ID ............................... 214
6.19.2.17 UCT_IFACE_FLAG_ERRHANDLE_REMOTE_MEM ....................... 215
6.19.2.18 UCT_IFACE_FLAG_ERRHANDLE_BCOPY_LEN .......................... 215
6.19.2.19 UCT_IFACE_FLAG_ERRHANDLE_PEER_FAILURE ..................... 215
6.19.2.20 UCT_IFACE_FLAG_EP_CHECK ......................................... 215
6.19.2.21 UCT_IFACE_FLAG_CONNECT_TO_IFACE ............................... 215
6.19.2.22 UCT_IFACE_FLAG_CONNECT_TO_EP ................................... 215
6.19.2.23 UCT_IFACE_FLAG_CONNECT_TO_SOCKADDR .......................... 215
6.19.2.24 UCT_IFACE_FLAG_AM_DUP .......................................... 216
6.19.2.25 UCT_IFACE_FLAG_CB_SYNC ......................................... 216
6.19.2.26 UCT_IFACE_FLAG_CB_ASYNC ........................................ 216
6.19.2.27 UCT_IFACE_FLAG_EP_KEEPALIVE ................................... 216
6.19.2.28 UCT_IFACE_FLAG_TAG_EAGER_SHORT ................................ 216
6.19.2.29 UCT_IFACE_FLAG_TAG_EAGER_BCOPY ............................... 216
6.19.2.30 UCT_IFACE_FLAG_TAG_EAGER_ZCOPY ................................ 216
6.19.2.31 UCT_IFACE_FLAG_TAG_RNDV_ZCOPY ................................ 216

6.20 UCT interface for asynchronous event capabilities .................. 217
   6.20.1 Detailed Description .............................................. 217
   6.20.2 Macro Definition Documentation .................................. 217
      6.20.2.1 UCT_IFACE_FLAG_EVENT_SEND_COMP .......................... 217
      6.20.2.2 UCT_IFACE_FLAG_EVENT_RECV ................................ 217
      6.20.2.3 UCT_IFACE_FLAG_EVENT_RECV_SIG ................................ 217
      6.20.2.4 UCT_IFACE_FLAG_EVENT_FD ................................... 217
      6.20.2.5 UCT_IFACE_FLAG_EVENT_ASYNC_CB ............................ 217

6.21 Unified Communication Services (UCS) API .......................... 218
   6.21.1 Detailed Description .............................................. 218

6.22 UCS Communication Resource ........................................... 219
   6.22.1 Detailed Description .............................................. 220
© 2021 Unified Communication X (UCX). All rights reserved.
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.22.2 Data Structure Documentation</td>
<td>220</td>
</tr>
<tr>
<td>6.22.2.1 struct ucs_sock_addr</td>
<td>220</td>
</tr>
<tr>
<td>6.22.2.2 Data Structure Documentation</td>
<td>220</td>
</tr>
<tr>
<td>6.22.3 Typedef Documentation</td>
<td>220</td>
</tr>
<tr>
<td>6.22.3.1 ucs_async_event_cb_t</td>
<td>220</td>
</tr>
<tr>
<td>6.22.3.2 ucs_sock_addr_t</td>
<td>220</td>
</tr>
<tr>
<td>6.22.3.3 ucs_memory_type_t</td>
<td>221</td>
</tr>
<tr>
<td>6.22.3.4 ucs_time_t</td>
<td>221</td>
</tr>
<tr>
<td>6.22.3.5 ucs_status_ptr_t</td>
<td>221</td>
</tr>
<tr>
<td>6.22.4 Enumeration Type Documentation</td>
<td>221</td>
</tr>
<tr>
<td>6.22.4.1 ucs_callbackq_flags</td>
<td>221</td>
</tr>
<tr>
<td>6.22.4.2 ucs_memory_type</td>
<td>221</td>
</tr>
<tr>
<td>6.22.4.3 ucs_status_t</td>
<td>222</td>
</tr>
<tr>
<td>6.22.4.4 ucs_thread_mode_t</td>
<td>223</td>
</tr>
<tr>
<td>6.22.5 Function Documentation</td>
<td>223</td>
</tr>
<tr>
<td>6.22.5.1 ucs_async_set_event_handler()</td>
<td>223</td>
</tr>
<tr>
<td>6.22.5.2 ucs_async_add_timer()</td>
<td>224</td>
</tr>
<tr>
<td>6.22.5.3 ucs_async_remove_handler()</td>
<td>224</td>
</tr>
<tr>
<td>6.22.5.4 ucs_async_modify_handler()</td>
<td>224</td>
</tr>
<tr>
<td>6.22.5.5 ucs_async_context_create()</td>
<td>225</td>
</tr>
<tr>
<td>6.22.5.6 ucs_async_context_destroy()</td>
<td>225</td>
</tr>
<tr>
<td>6.22.5.7 ucs_async_poll()</td>
<td>226</td>
</tr>
<tr>
<td>7 Data Structure Documentation</td>
<td>227</td>
</tr>
<tr>
<td>7.1 ucp_generic_dt_ops Struct Reference</td>
<td>227</td>
</tr>
<tr>
<td>7.1.1 Detailed Description</td>
<td>227</td>
</tr>
<tr>
<td>7.2 uct_tag_context Struct Reference</td>
<td>227</td>
</tr>
<tr>
<td>7.2.1 Detailed Description</td>
<td>228</td>
</tr>
<tr>
<td>7.2.2 Field Documentation</td>
<td>228</td>
</tr>
<tr>
<td>7.2.2.1 tag_consumed_cb</td>
<td>228</td>
</tr>
<tr>
<td>7.2.2.2 completed_cb</td>
<td>228</td>
</tr>
<tr>
<td>7.2.2.3 rndv_cb</td>
<td>229</td>
</tr>
<tr>
<td>7.2.2.4 priv</td>
<td>229</td>
</tr>
<tr>
<td>8 Example Documentation</td>
<td>231</td>
</tr>
<tr>
<td>8.1 ucp_client_server.c</td>
<td>231</td>
</tr>
<tr>
<td>8.2 ucp_hello_world.c</td>
<td>240</td>
</tr>
<tr>
<td>8.3 uct_hello_world.c</td>
<td>247</td>
</tr>
</tbody>
</table>

Index 255

© 2021 Unified Communication X (UCX). All rights reserved.
Chapter 1

Preface

1.1 Scope of the Document

This document describes the UCX programming interface. The programming interface exposes a high performance communication API, which provides basic building blocks for PGAS, Message Passing Interface (MPI), Big-Data, Analytics, File I/O, and storage library developers.

1.2 Audience

This manual is intended for programmers who want to develop parallel programming models like OpenSHMEM, MPI, UPC, Chapel, etc. The manual assumes that the reader is familiar with the following:

- Basic concepts of two-sided, one-sided, atomic, and collective operations
- C programming language

1.3 Document Status

This section briefly describes a list of open issues in the UCX specification.

- UCP API - work in progress
- UCT API - work in progress

1.4 License

UCX project follows open source development model and the software is licensed under BSD-3 license.
Chapter 2

Introduction

2.1 Motivation

A communication middleware abstracts the vendor-specific software and hardware interfaces. They bridge the semantic and functionality gap between the programming models and the software and hardware network interfaces by providing data transfer interfaces and implementation, optimized protocols for data transfer between various memories, and managing network resources. There are many communication middleware APIs and libraries to support parallel programming models such as MPI, OpenSHMEM, and task-based models.

Current communication middleware designs typically take two approaches. First, communication middleware such as Intel's PSM (previously Qlogic), Mellanox's MXM, and IBM's PAMI provide high-performance implementations for specific network hardware. Second, communication middleware such as VMI, Cactus, ARMCi, GASNet, and Open MPI are tightly coupled to a specific programming model. Communication middleware designed with either of this design approach requires significant porting effort to move a new network interface or programming model.

To achieve functional and performance portability across architectures and programming models, we introduce Unified Communication X (UCX).

2.2 UCX

Unified Communication X (UCX) is a set of network APIs and their implementations for high throughput computing. UCX is a combined effort of national laboratories, industry, and academia to design and implement a high-performing and highly-scalable network stack for next generation applications and systems. UCX design provides the ability to tailor its APIs and network functionality to suit a wide variety of application domains. We envision that these APIs will satisfy the networking needs of many programming models such as the Message Passing Interface (MPI), OpenSHMEM, Partitioned Global Address Space (PGAS) languages, task-based paradigms, and I/O bound applications.

The initial focus is on supporting semantics such as point-to-point communications (one-sided and two-sided), collective communication, and remote atomic operations required for popular parallel programming models. Also, the initial UCX reference implementation is targeted to support current network technologies such as:

- Open Fabrics - InfiniBand (Mellanox, Qlogic, IBM), libfabrics, iWARP, RoCE
- Cray GEMINI & ARIES
- Shared memory (MMAP, Posix, CMA, KNEM, XPMEM, etc.)
- Ethernet (TCP/UDP)

UCX design goals are focused on performance and scalability, while efficiently supporting popular and emerging programming models.
UCX’s API and design do not impose architectural constraints on the network hardware nor require any specific capabilities to the support the programming model functionality. This is achieved by keeping the API flexible and ability to support the missing functionality efficiently in the software.

Extreme scalability is an important design goal for UCX. To achieve this, UCX follows these design principles:

- Minimal memory consumption: Design avoids data-structures that scale with the number of processing elements (i.e., order N data structures), and share resources among multiple programming models.
- Low-latency Interfaces: Design provides at least two sets of APIs with one set focused on the performance, and the other focused on functionality.
- High bandwidth: With minimal software overhead combined and support for multi-rail and multi-device capabilities, the design provides all the hooks that are necessary for exploiting hardware bandwidth capabilities.
- Asynchronous Progress: API provides non-blocking communication interfaces and design supports asynchronous progress required for communication and computation overlap.
- Resilience: the API exposes communication control hooks required for fault tolerant communication library implementation.

UCX design provides native support for hybrid programming models. The design enables resource sharing, optimal memory usage, and progress engine coordination to efficiently implement hybrid programming models. For example, hybrid applications that use both OpenSHMEM and MPI programming models will be able to select between a single-shared UCX network context or a stand alone UCX network context for each one of them. Such flexibility, optimized resource sharing, and reduced memory consumption, improve network and application performance.
The UCX framework consists of the three main components: UC-Services (UCS), UC-Transports (UCT), and UC-Protocols (UCP). Each one of these components exports a public API, and can be used as a stand-alone library.

**Figure 3.1: UCX Framework Architecture**

### 3.1 UCS

UCS is a service layer that provides the necessary functionality for implementing portable and efficient utilities. This layer includes the following services:

- an abstraction for accessing platform specific functionality (atomic operations, thread safety, etc.),
- tools for efficient memory management (memory pools, memory allocators, and memory allocators hooks),
- commonly used data structures (hashes, trees, lists).

### 3.2 UCT

UCT is a transport layer that abstracts the differences across various hardware architectures and provides a low-level API that enables the implementation of communication protocols. The primary goal of the layer is to provide...
direct and efficient access to hardware network functionality. For this purpose, UCT relies on vendor provided low-level drivers such as InfiniBand Verbs, Cray’s uGNI, libfabrics, etc. In addition, the layer provides constructs for communication context management (thread-based and application level), and allocation and management of device-specific memories including those found in accelerators. In terms of communication APIs, UCT defines interfaces for immediate (short), buffered copy-and-send (bcopy), and zero-copy (zcopy) communication operations.

**Short:** This type of operation is optimized for small messages that can be posted and completed in place.

**Bcopy:** This type of operation is optimized for medium size messages that are typically sent through a so-called bouncing-buffer. This auxiliary buffer is typically allocated given network constraints and ready for immediate utilization by the hardware. Since a custom data packing routine could be provided, this method can be used for non-contiguous i/o.

**Zcopy:** This type of operation exposes zero-copy memory-to-memory communication semantics, which means that message is sent directly from user buffer, or received directly to user buffer, without being copied between the network layers.

### 3.3 UCP

UCP implements higher-level protocols that are typically used by message passing (MPI) and PGAS programming models by using lower-level capabilities exposed through the UCT layer. UCP is provides the following functionality: ability to select different transports for communication, message fragmentation, multi-rail communication, and initializing and finalizing the library. Currently, the API has the following classes of interfaces: Initialization, Remote Memory Access (RMA) communication, Atomic Memory Operations (AMO), Active Message, Tag-Matching, and Collectives.

**Initialization:** This subset of interfaces defines the communication context setup, queries the network capabilities, and initializes the local communication endpoints. The context represented by the UCX context is an abstraction of the network transport resources. The communication endpoint setup interfaces initialize the UCP endpoint, which is an abstraction of all the necessary resources associated with a particular connection. The communication endpoints are used as input to all communication operations to describe the source and destination of the communication.

**RMA:** This subset of interfaces defines one-sided communication operations such as PUT and GET, required for implementing low overhead, direct memory access communications constructs needed by both distributed and shared memory programming models. UCP includes a separate set of interfaces for communicating non-contiguous data. This functionality was included to support various programming models' communication requirements and leverage the scatter/gather capabilities of modern network hardware.

**AMO:** This subset of interfaces provides support for atomically performing operations on the remote memory, an important class of operations for PGAS programming models, particularly OpenSHMEM.

**Tag Matching:** This interface supports tag-matching for send-receive semantics which is a key communication semantic defined by the MPI specification.

**Stream:** The API provides order and reliable communication semantics. Data is treated as an ordered sequence of bytes pushed through the connection. In contrast of tag-matching interface, the size of each individual send does not necessarily have to match the size of each individual receive, as long as the total number of bytes is the same. This API is designed to match widely used BSD-socket based programming models.

**Active Message:** A subset of functionality where the incoming packet invokes a sender-specified callback in order to be processed by the receiving process. As an example, the two-sided MPI interface can easily be implemented on top of such a concept (TBD: cite openmpi ). However, these interfaces are more general and suited for other programming paradigms where the receiver process does not prepost receives, but expects to react to incoming packets directly. Like RMA and tag-matching interfaces, the active message interface provides separate APIs for different message types and non-contiguous data.

**Collectives:** This subset of interfaces defines group communication and synchronization operations. The collective operations include barrier, all-to-one, all-to-all, and reduction operations. When possible, we will take advantage of hardware acceleration for collectives (e.g., InfiniBand Switch collective acceleration).
Chapter 4

Conventions and Notations

This section describes the conventions and notations in the UCX specification.

4.1 Blocking Behavior

The blocking UCX routines return only when a UCX operation is complete. After the return, the resources used in the UCX routine are available for reuse.

4.2 Non-blocking Behavior

The non-blocking UCX routines return immediately, independent of operation completion. After the return, the resources used for the routines are not necessarily available for reuse.

4.3 Fairness

UCX routines do not guarantee fairness. However, the routines enable UCX consumers to write efficient and fair programs.

4.4 Interaction with Signal Handler Functions

If UCX routines are invoked from a signal handler function, the behavior of the program is undefined.
Chapter 5

Deprecated List

Global `ucp_atomic_add32` (uctp_ep_h ep, uint32_t add, uint64_t remote_addr, ucp_rkey_h rkey)
  Replaced by `ucp_atomic_post` with opcode UCP_ATOMIC_POST_OP_ADD.

Global `ucp_atomic_add64` (uctp_ep_h ep, uint64_t add, uint64_t remote_addr, ucp_rkey_h rkey)
  Replaced by `ucp_atomic_post` with opcode UCP_ATOMIC_POST_OP_ADD.

Global `ucp_atomic_cswap32` (uctp_ep_h ep, uint32_t compare, uint32_t swap, uint64_t remote_addr, ucp_rkey_h rkey, uint32_t *result)
  Replaced by `ucp_atomic_fetch_nb` with opcode UCP_ATOMIC_FETCH_OP_CSWAP.

Global `ucp_atomic_cswap64` (uctp_ep_h ep, uint64_t compare, uint64_t swap, uint64_t remote_addr, ucp_rkey_h rkey, uint64_t *result)
  Replaced by `ucp_atomic_fetch_nb` with opcode UCP_ATOMIC_FETCH_OP_CSWAP.

Global `ucp_atomic_fadd32` (uctp_ep_h ep, uint32_t add, uint64_t remote_addr, ucp_rkey_h rkey, uint32_t *result)
  Replaced by `ucp_atomic_fetch_nb` with opcode UCP_ATOMIC_FETCH_OP_FADD.

Global `ucp_atomic_fadd64` (uctp_ep_h ep, uint64_t add, uint64_t remote_addr, ucp_rkey_h rkey, uint64_t *result)
  Replaced by `ucp_atomic_fetch_nb` with opcode UCP_ATOMIC_FETCH_OP_FADD.

Global `ucp_atomic_swap32` (uctp_ep_h ep, uint32_t swap, uint64_t remote_addr, ucp_rkey_h rkey, uint32_t *result)
  Replaced by `ucp_atomic_fetch_nb` with opcode UCP_ATOMIC_FETCH_OP_SWAP.

Global `ucp_atomic_swap64` (uctp_ep_h ep, uint64_t swap, uint64_t remote_addr, ucp_rkey_h rkey, uint64_t *result)
  Replaced by `ucp_atomic_fetch_nb` with opcode UCP_ATOMIC_FETCH_OP_SWAP.

Global `ucp_disconnect_nb` (uctp_ep_h ep)
  Replaced by `ucp_ep_close_nb`.

Global `ucp_ep_destroy` (uctp_ep_h ep)
  Replaced by `ucp_ep_close_nb`.

Global `ucp_ep_flush` (uctp_ep_h ep)
  Replaced by `ucp_ep_flush_nb`.

Global `ucp_ep_modify_nb` (uctp_ep_h ep, const ucp_ep_params_t *params)
  Use `ucp_listener_conn_handler_t` instead of `ucp_listener_accept_handler_t`, if you have other use cases please submit an issue on https://github.com/openucx/ucx or report to ucx-group@elist.ornl.gov

Global `ucp_get` (uctp_ep_h ep, void *buffer, size_t length, uint64_t remote_addr, ucp_rkey_h rkey)
  Replaced by `ucp_get_nb`. 
Global `ucp_listener_accept_handler_t`
   Replaced by `ucp_listener_conn_handler_t`.

Global `ucp_listener_accept_handler_t`
   Replaced by `ucp_listener_conn_handler_t`.

Global `ucp_put (ucp_ep_h ep, const void *buffer, size_t length, uint64_t remote_addr, ucp_rkey_h rkey)`
   Replaced by `ucp_put_nb`. The following example implements the same functionality using `ucp_put_nb`:

Global `ucp_request_is_completed (void *request)`
   Replaced by `ucp_request_test`.

Global `ucp_request_release (void *request)`
   Replaced by `ucp_request_free`.

Global `ucp_request_test (void *request, ucp_tag_recv_info_t *info)`
   Replaced by `ucp_tag_recv_request_test` and `ucp_request_check_status` depends on use case.

Global `ucp_worker_flush (ucp_worker_h worker)`
   Replaced by `ucp_worker_flush_nb`. The following example implements the same functionality using `ucp_worker_flush_nb`:
Chapter 6

Module Documentation

6.1 Unified Communication Protocol (UCP) API

Modules

• UCP Application Context
• UCP Worker
• UCP Memory routines
• UCP Wake-up routines
• UCP Endpoint
• UCP Communication routines
• UCP Configuration
• UCP Data type routines

6.1.1 Detailed Description

This section describes UCP API.
6.2 UCP Application Context

Data Structures

- struct ucp_context_attr
  Context attributes. More...
- struct ucp_tag_recv_info
  UCP receive information descriptor. More...
- struct ucp_request_param_t
  Operation parameters passed to ucp_tag_send_nbx, ucp_tag_send_sync_nbx, ucp_tag_recv_nbx, ucp_put_nbx, ucp_get_nbx, ucp_am_send_nbx and ucp_am_recv_data_nbx. More...
- union ucp_request_param_t.cb
- union ucp_request_param_t.recv_info

Typedefs

- typedef struct ucp_context_attr ucp_context_attr_t
  Context attributes.
- typedef struct ucp_tag_recv_info ucp_tag_recv_info_t
  UCP receive information descriptor.
- typedef struct ucp_context * ucp_context_h
  UCP Application Context.
- typedef void (∗ ucp_request_init_callback_t) (void ∗ request)
  Request initialization callback.
- typedef void (∗ ucp_request_cleanup_callback_t) (void ∗ request)
  Request cleanup callback.

Enumerations

- enum ucp_params_field {
  UCP_PARAM_FIELD_FEATURES = UCS_BIT(0), UCP_PARAM_FIELD_REQUEST_SIZE = UCS_BIT(1),
  UCP_PARAM_FIELD_REQUEST_INIT = UCS_BIT(2), UCP_PARAM_FIELD_REQUEST_CLEANUP = UCS_BIT(3),
  UCP_PARAM_FIELD_TAG_SENDER_MASK = UCS_BIT(4), UCP_PARAM_FIELD_MT_WORKERS_SHARED = UCS_BIT(5),
  UCP_PARAM_FIELD_ESTIMATED_NUM_EPS = UCS_BIT(6), UCP_PARAM_FIELD_ESTIMATED_NUM_PPN = UCS_BIT(7) }
  UCP context parameters field mask.
- enum ucp_feature {
  UCP_FEATURE_TAG = UCS_BIT(0), UCP_FEATURE_RMA = UCS_BIT(1), UCP_FEATURE_AMO32 = UCS_BIT(2),
  UCP_FEATURE_AMO64 = UCS_BIT(3), UCP_FEATURE_WAKEUP = UCS_BIT(4), UCP_FEATURE_STREAM = UCS_BIT(5),
  UCP_FEATURE_AM = UCS_BIT(6) }
  UCP configuration features.
- enum ucp_context_attr_field {
  UCP_ATTR_FIELD_REQUEST_SIZE = UCS_BIT(0), UCP_ATTR_FIELD_THREAD_MODE = UCS_BIT(1),
  UCP_ATTR_FIELD_MEMORY_TYPES = UCS_BIT(2) }
  UCP context attributes field mask.

Functions

- void ucp_get_version (unsigned ∗ major_version, unsigned ∗ minor_version, unsigned ∗ release_number)
  Get UCP library version.
- const char ∗ ucp_get_version_string (void)
6.2 UCP Application Context

Get UCP library version as a string.

- static ucs_status_t ucp_init (const ucp_params_t *params, const ucp_config_t *config, ucp_context_h *context_p)
  UCP context initialization.

- void ucp_cleanup (ucp_context_h context_p)
  Release UCP application context.

- ucs_status_t ucp_context_query (ucp_context_h context_p, ucp_context_attr_t *attr)
  Get attributes specific to a particular context.

- void ucp_context_print_info (const ucp_context_h context, FILE *stream)
  Print context information.

6.2.1 Detailed Description

Application context is a primary concept of UCP design which provides an isolation mechanism, allowing resources associated with the context to separate or share network communication context across multiple instances of applications.

This section provides a detailed description of this concept and routines associated with it.

6.2.2 Data Structure Documentation

6.2.2.1 struct ucp_context_attr

The structure defines the attributes which characterize the particular context.

Data Fields

<table>
<thead>
<tr>
<th>Field Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint64_t</td>
<td>field_mask</td>
<td>Mask of valid fields in this structure, using bits from ucp_context_attr_field. Fields not specified in this mask will be ignored. Provides ABI compatibility with respect to adding new fields.</td>
</tr>
<tr>
<td>size_t</td>
<td>request_size</td>
<td>Size of UCP non-blocking request. When pre-allocated request is used (e.g. in ucp_tag_recv_nbr) it should have enough space to fit UCP request data, which is defined by this value.</td>
</tr>
<tr>
<td>ucs_thread_mode_t</td>
<td>thread_mode</td>
<td>Thread safe level of the context. For supported thread levels please see ucs_thread_mode_t.</td>
</tr>
<tr>
<td>uint64_t</td>
<td>memory_types</td>
<td>Mask of which memory types are supported, for supported memory types please see ucs_memory_type_t.</td>
</tr>
</tbody>
</table>

6.2.2.2 struct ucp_tag_recv_info

The UCP receive information descriptor is allocated by application and filled in with the information about the received message by ucp_tag_probe_nb or ucp_tag_recv_request_test routines or ucp_tag_recv_callback_t callback argument.

Examples

ucp_client_server.c, and ucp_hello_world.c.

Data Fields

<table>
<thead>
<tr>
<th>Field Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ucp_tag_t</td>
<td>sender_tag</td>
<td>Sender tag</td>
</tr>
<tr>
<td>size_t</td>
<td>length</td>
<td>The size of the received data</td>
</tr>
</tbody>
</table>
6.2.2.3 struct ucp_request_param_t

The structure `ucp_request_param_t` is used to specify datatype of operation, provide user request in case the external request is used, set completion callback and custom user data passed to this callback.

Example: implementation of function to send contiguous buffer to ep and invoke callback function at operation completion. If the operation completed immediately (status == UCS_OK) then callback is not called.

```c
ucs_status_ptr_t send_data(ucp_ep_h ep, void *buffer, size_t length, ucp_tag_t tag, void *request)
{
    ucp_request_param_t param = {
        .op_attr_mask = UCP_OP_ATTR_FIELD_CALLBACK | UCP_OP_ATTR_FIELD_REQUEST,
        .request = request,
        .cb.ucp_send_nbx_callback_t = custom_send_callback_f,
        .user_data = pointer_to_user_context_passed_to_cb,
    };
    ucs_status_ptr_t status;
    status = ucp_tag_send_nbx(ep, buffer, length, tag, &param);
    if (UCS_PTR_IS_ERR(status)) {
        handle_error(status);
    } else if (status == UCS_OK) {
        // operation is completed
    }
    return status;
}
```

Examples

ucp_client_server.c, and ucp_hello_world.c.

Data Fields

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint32_t</td>
<td><code>op_attr_mask</code> Mask of valid fields in this structure and operation flags, using bits from <code>ucp_op_attr_t</code>. Fields not specified in this mask will be ignored. Provides ABI compatibility with respect to adding new fields.</td>
</tr>
<tr>
<td>uint32_t</td>
<td><code>flags</code> Request handle allocated by the user. There should be at least UCP request size bytes of available space before the request. The size of the UCP request can be obtained by <code>ucp_context_query</code> function.</td>
</tr>
<tr>
<td>void *</td>
<td><code>request</code> Callback function that is invoked whenever the send or receive operation is completed.</td>
</tr>
<tr>
<td>ucp_datatype_t</td>
<td><code>datatype</code> Datatype descriptor for the elements in the buffer. In case the <code>op_attr_mask</code> &amp; <code>UCP_OP_ATTR_FIELD_DATATYPE</code> bit is not set, then use default datatype <code>ucp_dt_make_contig(1)</code></td>
</tr>
<tr>
<td>void *</td>
<td><code>user_data</code> Pointer to user data passed to callback function.</td>
</tr>
<tr>
<td>void *</td>
<td><code>reply_buffer</code> Reply buffer. Can be used for storing operation result, for example by <code>ucp_atomic_op_nbx</code>.</td>
</tr>
<tr>
<td>ucs_memory_type_t</td>
<td><code>memory_type</code> Memory type of the buffer. see <code>ucs_memory_type_t</code> for possible memory types. An optimization hint to avoid memory type detection for request buffer. If this value is not set (along with its corresponding bit in the <code>op_attr_mask</code> - <code>UCP_OP_ATTR_FIELD_MEMORY_TYPE</code>) then use default <code>UCS_MEMORY_TYPE_UNKNOWN</code> which means the memory type will be detected internally.</td>
</tr>
<tr>
<td>union ucp_request_param_t</td>
<td><code>recv_info</code> Pointer to the information where received data details are stored in case of an immediate completion of receive operation. The user has to provide a pointer to valid memory/variable which will be updated on function return.</td>
</tr>
</tbody>
</table>
6.2 UCP Application Context

6.2.2.4 union ucp_request_param_t.cb

Callback function that is invoked whenever the send or receive operation is completed.

Data Fields

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ucp_send_nbx_callback_t</td>
<td>send</td>
</tr>
<tr>
<td>ucp_tag_recv_nbx_callback_t</td>
<td>recv</td>
</tr>
<tr>
<td>ucp_stream_recv_nbx_callback_t</td>
<td>recv_stream</td>
</tr>
<tr>
<td>ucp_am_recv_data_nbx_callback_t</td>
<td>recv_am</td>
</tr>
</tbody>
</table>

6.2.2.5 union ucp_request_param_t.recv_info

Pointer to the information where received data details are stored in case of an immediate completion of receive operation. The user has to provide a pointer to valid memory/variable which will be updated on function return.

Data Fields

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size_t *</td>
<td>length</td>
</tr>
<tr>
<td>ucp_tag_recv_info_t *</td>
<td>tag_info</td>
</tr>
</tbody>
</table>

6.2.3 Typedef Documentation

6.2.3.1 ucp_context_attr_t

typedef struct ucp_context_attr ucp_context_attr_t

The structure defines the attributes which characterize the particular context.

6.2.3.2 ucp_tag_recv_info_t

typedef struct ucp_tag_recv_info ucp_tag_recv_info_t

The UCP receive information descriptor is allocated by application and filled in with the information about the received message by ucp_tag_probe_nb or ucp_tag_recv_request_test routines or ucp_tag_recv_callback_t callback argument.

6.2.3.3 ucp_context_h

typedef struct ucp_context* ucp_context_h

UCP application context (or just a context) is an opaque handle that holds a UCP communication instance’s global information. It represents a single UCP communication instance. The communication instance could be an OS process (an application) that uses UCP library. This global information includes communication resources, endpoints, memory, temporary file storage, and other communication information directly associated with a specific UCP instance. The context also acts as an isolation mechanism, allowing resources associated with the context to manage multiple concurrent communication instances. For example, users using both MPI and OpenSHMEM sessions simultaneously can isolate their communication by allocating and using separate contexts for each of them. Alternatively, users can share the communication resources (memory, network resource context, etc.) between them by using the same application context. A message sent or a RMA operation performed in one application context cannot be received in any other application context.

© 2021 Unified Communication X (UCX). All rights reserved.
6.2.3.4 ucp_request_init_callback_t

typedef void(* ucp_request_init_callback_t) (void *request)

This callback routine is responsible for the request initialization.

Parameters

- in request Request handle to initialize.

6.2.3.5 ucp_request_cleanup_callback_t

typedef void(* ucp_request_cleanup_callback_t) (void *request)

This callback routine is responsible for cleanup of the memory associated with the request.

Parameters

- in request Request handle to cleanup.

6.2.4 Enumeration Type Documentation

6.2.4.1 ucp_params_field

enum ucp_params_field

The enumeration allows specifying which fields in ucp_params_t are present. It is used to enable backward compatibility support.

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCP_PARAM_FIELD_FEATURES</td>
<td>features</td>
</tr>
<tr>
<td>UCP_PARAM_FIELD_REQUEST_SIZE</td>
<td>request_size</td>
</tr>
<tr>
<td>UCP_PARAM_FIELD_REQUEST_INIT</td>
<td>request_init</td>
</tr>
<tr>
<td>UCP_PARAM_FIELD_REQUEST_CLEANUP</td>
<td>request_cleanup</td>
</tr>
<tr>
<td>UCP_PARAM_FIELD_TAG_SENDER_MASK</td>
<td>tag_sender_mask</td>
</tr>
<tr>
<td>UCP_PARAM_FIELD_MT_WORKERS_SHARED</td>
<td>mt_workers_shared</td>
</tr>
<tr>
<td>UCP_PARAM_FIELD_ESTIMATED_NUM_EPS</td>
<td>estimated_num_eps</td>
</tr>
<tr>
<td>UCP_PARAM_FIELD_ESTIMATED_NUM_PPN</td>
<td>estimated_num_ppn</td>
</tr>
</tbody>
</table>

6.2.4.2 ucp_feature

enum ucp_feature

The enumeration list describes the features supported by UCP. An application can request the features using UCP parameters during UCP initialization process.
6.2 UCP Application Context

Enumerator

<table>
<thead>
<tr>
<th>Enumerators</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCP_FEATURE_TAG</td>
<td>Request tag matching support</td>
</tr>
<tr>
<td>UCP_FEATURE_RMA</td>
<td>Request remote memory access support</td>
</tr>
<tr>
<td>UCP_FEATURE_AMO32</td>
<td>Request 32-bit atomic operations support</td>
</tr>
<tr>
<td>UCP_FEATURE_AMO64</td>
<td>Request 64-bit atomic operations support</td>
</tr>
<tr>
<td>UCP_FEATURE_WAKEUP</td>
<td>Request interrupt notification support</td>
</tr>
<tr>
<td>UCP_FEATURE_STREAM</td>
<td>Request stream support</td>
</tr>
<tr>
<td>UCP_FEATURE_AM</td>
<td>Request Active Message support</td>
</tr>
</tbody>
</table>

6.2.4.3 ucp_context_attr_field

enum ucp_context_attr_field

The enumeration allows specifying which fields in ucp_context_attr_t are present. It is used to enable backward compatibility support.

Enumerator

<table>
<thead>
<tr>
<th>Enumerators</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCP_ATTR_FIELD_REQUEST_SIZE</td>
<td>UCP request size</td>
</tr>
<tr>
<td>UCP_ATTR_FIELD_THREAD_MODE</td>
<td>UCP context thread flag</td>
</tr>
<tr>
<td>UCP_ATTR_FIELD_MEMORY_TYPES</td>
<td>UCP supported memory types</td>
</tr>
</tbody>
</table>

6.2.5 Function Documentation

6.2.5.1 ucp_get_version()

void ucp_get_version ( 
    unsigned * major_version, 
    unsigned * minor_version, 
    unsigned * release_number )

This routine returns the UCP library version.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>major_version</td>
<td>Filled with library major version.</td>
</tr>
<tr>
<td>minor_version</td>
<td>Filled with library minor version.</td>
</tr>
<tr>
<td>release_number</td>
<td>Filled with library release number.</td>
</tr>
</tbody>
</table>

6.2.5.2 ucp_get_version_string()

const char* ucp_get_version_string ( 
    void )

This routine returns the UCP library version as a string which consists of: "major.minor.release".

© 2021 Unified Communication X (UCX). All rights reserved.
6.2.5.3 ucp_init()

```c
static ucs_status_t ucp_init {
    const ucp_params_t * params,
    const ucp_config_t * config,
    ucp_context_h * context_p } [inline], [static]
```

This routine creates and initializes a UCP application context.

**Warning**

This routine must be called before any other UCP function call in the application.

This routine checks API version compatibility, then discovers the available network interfaces, and initializes the network resources required for discovering of the network and memory related devices. This routine is responsible for initialization all information required for a particular application scope, for example, MPI application, OpenSHMEM application, etc.

**Note**

- Higher level protocols can add additional communication isolation, as MPI does with it's communicator object. A single communication context may be used to support multiple MPI communicators.
- The context can be used to isolate the communication that corresponds to different protocols. For example, if MPI and OpenSHMEM are using UCP to isolate the MPI communication from the OpenSHMEM communication, users should use different application context for each of the communication libraries.

**Parameters**

<table>
<thead>
<tr>
<th>in</th>
<th>config</th>
<th>UCP configuration descriptor allocated through ucp_config_read() routine.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>params</td>
<td>User defined ucp_params_t configurations for the UCP application context.</td>
</tr>
<tr>
<td>out</td>
<td>context_p</td>
<td>Initialized UCP application context.</td>
</tr>
</tbody>
</table>

**Returns**

Error code as defined by ucs_status_t

**Examples**

ucp_client_server.c, and ucp_hello_world.c.

6.2.5.4 ucp_cleanup()

```c
void ucp_cleanup {
    ucp_context_h context_p }
```

This routine finalizes and releases the resources associated with a UCP application context.

**Warning**

An application cannot call any UCP routine once the UCP application context released.

The cleanup process releases and shuts down all resources associated with the application context. After calling this routine, calling any UCP routine without calling **UCP initialization routine** is invalid.
6.2 UCP Application Context

Parameters

| in | context_p | Handle to UCP application context. |

Examples

ucp_client_server.c, and ucp_hello_world.c.

6.2.5.5 ucp_context_query()

```c
ucs_status_t ucp_context_query(
    ucp_context_h context_p,
    ucp_context_attr_t *attr)
```

This routine fetches information about the context.

Parameters

| in | context_p | Handle to UCP application context. |
| out | attr | Filled with attributes of context_p context. |

Returns

Error code as defined by ucs_status_t

6.2.5.6 ucp_context_print_info()

```c
void ucp_context_print_info(
    const ucp_context_h context,
    FILE *stream)
```

This routine prints information about the context configuration: including memory domains, transport resources, and other useful information associated with the context.

Parameters

| in | context | Print this context object's configuration. |
| in | stream | Output stream on which to print the information. |
6.3 UCP Worker

Data Structures

- **struct ucp_worker_attr**
  UCP worker attributes. More...
- **struct ucp_worker_params**
  Tuning parameters for the UCP worker. More...
- **struct ucp_listener_attr**
  UCP listener attributes. More...
- **struct ucp_conn_request_attr**
  UCP listener's connection request attributes. More...
- **struct ucp_listener_params**
  Parameters for a UCP listener object. More...
- **struct ucp_am_handler_param**
  Active Message handler parameters passed to ucp_worker_set_am_recv_handler routine. More...
- **struct ucp_am_recv_param**
  Operation parameters provided in ucp_am_recv_callback_t callback. More...
- **struct ucp_listener_accept_handler**
- **struct ucp_listener_conn_handler**
  UCP callback to handle the connection request in a client-server connection establishment flow. More...

Typedefs

- **typedef struct ucp_worker_attr ucp_worker_attr_t**
  UCP worker attributes.
- **typedef struct ucp_worker_params ucp_worker_params_t**
  Tuning parameters for the UCP worker.
- **typedef struct ucp_listener_attr ucp_listener_attr_t**
  UCP listener attributes.
- **typedef struct ucp_conn_request_attr ucp_conn_request_attr_t**
  UCP listener's connection request attributes.
- **typedef struct ucp_listener_params ucp_listener_params_t**
  Parameters for a UCP listener object.
- **typedef struct ucp_am_handler_param ucp_am_handler_param_t**
  Active Message handler parameters passed to ucp_worker_set_am_recv_handler routine.
- **typedef struct ucp_am_recv_param ucp_am_recv_param_t**
  Operation parameters provided in ucp_am_recv_callback_t callback.
- **typedef struct ucp_address ucp_address_t**
  UCP worker address.
- **typedef struct ucp_listener ucp_listener_h**
  UCP listen handle.
- **typedef struct ucp_worker ucp_worker_h**
  UCP Worker.
- **typedef void(uccp_listener_accept_callback_t) (ucp_ep_h ep, void *arg)**
  A callback for accepting client/server connections on a listener ucp_listener_h.
- **typedef void(uccp_listener_conn_callback_t) (ucp_conn_request_h conn_request, void *arg)**
  A callback for handling of incoming connection request conn_request from a client.
- **typedef struct ucp_listener_conn_handler ucp_listener_conn_handler_t**
  UCP callback to handle the connection request in a client-server connection establishment flow.
- **typedef enum ucp_wakeup_event_types ucp_wakeup_event_t**
  UCP worker wakeup events mask.
Enumerations

- enum ucp_worker_params_field {
  UCP_WORKER_PARAM_FIELD_THREAD_MODE = UCS_BIT(0),
  UCP_WORKER_PARAM_FIELD_CPU_MASK
  = UCS_BIT(1), UCP_WORKER_PARAM_FIELD_EVENTS = UCS_BIT(2),
  UCP_WORKER_PARAM_FIELD_USER_DATA
  = UCS_BIT(3),
  UCP_WORKER_PARAM_FIELD_EVENT_FD = UCS_BIT(4) }

  UCP worker parameters field mask.

- enum ucp_listener_params_field {
  UCP_LISTENER_PARAM_FIELD_SOCK_ADDR = UCS_BIT(0),
  UCP_LISTENER_PARAM_FIELD_ACCEPT_HANDLER = UCS_BIT(1),
  UCP_LISTENER_PARAM_FIELD_CONN_HANDLER
  = UCS_BIT(2) }

  UCP listener parameters field mask.

- enum ucp_worker_address_flags_t {
  UCP_WORKER_ADDRESS_FLAG_NET_ONLY = UCS_BIT(0) }

  UCP worker address flags.

- enum ucp_worker_attr_field {
  UCP_WORKER_ATTR_FIELD_THREAD_MODE = UCS_BIT(0),
  UCP_WORKER_ATTR_FIELD_ADDRESS
  = UCS_BIT(1), UCP_WORKER_ATTR_FIELD_ADDRESS_FLAGS
  = UCS_BIT(2), UCP_WORKER_ATTR_FIELD_MAX_AM_HEADER
  = UCS_BIT(3) }

  UCP worker attributes field mask.

- enum ucp_listener_attr_field {
  UCP_LISTENER_ATTR_FIELD_SOCKADDR = UCS_BIT(0) }

  UCP listener attributes field mask.

- enum ucp_conn_request_attr_field {
  UCP_CONN_REQUEST_ATTR_FIELD_CLIENT_ADDR
  = UCS_BIT(0) }

  UCP listener's connection request attributes field mask.

- enum ucp_am_cb_flags { UCP_AM_FLAG_WHOLE_MSG = UCS_BIT(0) }

  Flags for a UCP Active Message callback.

- enum ucp_send_am_flags { UCP_AM_SEND_FLAG_REPLY = UCS_BIT(0),
  UCP_AM_SEND_FLAG_EAGER
  = UCS_BIT(1), UCP_AM_SEND_FLAG_RNDV
  = UCS_BIT(2), UCP_AM_SEND_REPLY
  = UCP_AM_SEND_FLAG(1) }

  Flags for sending a UCP Active Message.

- enum ucp_wakeup_event_types { UCP_WAKEUP_RMA = UCS_BIT(0),
  UCP_WAKEUP_AMO = UCS_BIT(1), UCP_WAKEUP_TAG_SEND
  = UCS_BIT(2), UCP_WAKEUP_TAG_RECV
  = UCS_BIT(3), UCP_WAKEUP_TX
  = UCS_BIT(10), UCP_WAKEUP_RX
  = UCS_BIT(11), UCP_WAKEUP_EDGE
  = UCS_BIT(16) }

  UCP worker wakeup events mask.

Functions

- ucs_status_t ucp_worker_create (ucp_context_h context, const ucp_worker_params_t *params, ucp_worker_h *
  worker_p)

  Create a worker object.

- void ucp_worker_destroy (ucp_worker_h worker)

  Destroy a worker object.

- ucs_status_t ucp_worker_query (ucp_worker_h worker, ucp_worker_attr_t *attr)

  Get attributes specific to a particular worker.

- void ucp_worker_print_info (ucp_worker_h worker, FILE *stream)

  Print information about the worker.

- ucs_status_t ucp_worker_get_address (ucp_worker_h worker, ucp_address_t *address_p, size_t *
  address_length_p)

  Get the address of the worker object.

- void ucp_worker_release_address (ucp_worker_h worker, ucp_address_t *address)

  Release an address of the worker object.

- unsigned ucp_worker_progress (ucp_worker_h worker)
Progress all communications on a specific worker.

- `ssize_t ucp_stream_worker_poll (ucp_worker_h worker, ucp_stream_poll_ep_t *poll_eps, size_t max_eps, unsigned flags)`
  
  Poll for endpoints that are ready to consume streaming data.

- `ucs_status_t ucp_listener_create (ucp_worker_h worker, const ucp_listener_params_t *params, ucp_listener_h *listener_p)`
  
  Accept connections on a local address of the worker object.

- `void ucp_listener_destroy (ucp_listener_h listener)`
  
  Stop accepting connections on a local address of the worker object.

- `ucs_status_t ucp_listener_query (ucp_listener_h listener, ucp_listener_attr_t *attr)`
  
  Get attributes specific to a particular listener.

- `ucs_status_t ucp_conn_request_query (ucp_conn_request_h conn_request, ucp_conn_request_attr_t *attr)`
  
  Get attributes specific to a particular connection request received on the server side.

- `ucs_status_t ucp_listener_reject (ucp_listener_h listener, ucp_conn_request_h conn_request)`
  
  Reject an incoming connection request.

- `ucs_status_t ucp_worker_set_am_handler (ucp_worker_h worker, uint16_t id, ucp_am_callback_t cb, void *arg, uint32_t flags)`
  
  Add user defined callback for Active Message.

- `ucs_status_t ucp_worker_set_am_recv_handler (ucp_worker_h worker, const ucp_am_handler_param_t *param)`
  
  Add user defined callback for Active Message.

- `ucs_status_t ucp_worker_fence (ucp_worker_h worker)`
  
  Assures ordering between non-blocking operations.

- `ucs_status_ptr_t ucp_worker_flush_nb (ucp_worker_h worker, unsigned flags, ucp_send_callback_t cb)`
  
  Flush outstanding AMO and RMA operations on the worker.

- `ucs_status_ptr_t ucp_worker_flush_nbx (ucp_worker_h worker, const ucp_request_param_t *param)`
  
  Flush outstanding AMO and RMA operations on the worker.

- `ucs_status_t ucp_worker_flush (ucp_worker_h worker)`
  
  Flush outstanding AMO and RMA operations on the worker.

### 6.3.1 Detailed Description

UCP Worker routines

### 6.3.2 Data Structure Documentation

#### 6.3.2.1 struct ucp_worker_attr

The structure defines the attributes which characterize the particular worker.

**Data Fields**

<table>
<thead>
<tr>
<th>Type</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uint64_t</code></td>
<td><code>field_mask</code></td>
<td>Mask of valid fields in this structure, using bits from <code>ucp_worker_attr_field</code>. Fields not specified in this mask will be ignored. Provides ABI compatibility with respect to adding new fields.</td>
</tr>
<tr>
<td><code>ucs_thread_mode_t</code></td>
<td><code>thread_mode</code></td>
<td>Thread safe level of the worker.</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>address_flags</code></td>
<td>Flags indicating requested details of the worker address. If <code>UCP_WORKER_ATTR_FIELD_ADDRESS_FLAGS</code> bit is set in the field_mask, this value should be set as well. Possible flags are specified in <code>ucp_worker_address_flags_t</code>. Note This is an input attribute.</td>
</tr>
</tbody>
</table>
6.3 UCP Worker

Data Fields

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ucp_address_t</td>
<td>address</td>
<td>Worker address, which can be passed to remote instances of the UCP library to connect to this worker. The memory for the address handle is allocated by <code>ucp_worker_query()</code> routine, and must be released by using <code>ucp_worker_release_address()</code> routine.</td>
</tr>
<tr>
<td>size_t</td>
<td>address_length</td>
<td>Size of worker address in bytes.</td>
</tr>
<tr>
<td>size_t</td>
<td>max_am_header</td>
<td>Maximal allowed header size for <code>ucp_am_send_nbx</code> routine</td>
</tr>
</tbody>
</table>

6.3.2.2 struct ucp_worker_params

The structure defines the parameters that are used for the UCP worker tuning during the UCP worker creation.

Examples

`ucp_client_server.c`, and `ucp_hello_world.c`.

Data Fields

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint64_t</td>
<td>field_mask</td>
<td>Mask of valid fields in this structure, using bits from <code>ucp_worker_params_field</code>. Fields not specified in this mask will be ignored. Provides ABI compatibility with respect to adding new fields.</td>
</tr>
<tr>
<td>ucs_thread_mode_t</td>
<td>thread_mode</td>
<td>The parameter thread_mode suggests the thread safety mode which worker and the associated resources should be created with. This is an optional parameter. The default value is UCS_THREAD_MODE_SINGLE and it is used when the value of the parameter is not set. When this parameter along with its corresponding bit in the field_mask - UCP_WORKER_PARAM_FIELD_THREAD_MODE is set, the <code>ucp_worker_create</code> attempts to create worker with this thread mode. The thread mode with which worker is created can differ from the suggested mode. The actual thread mode of the worker should be obtained using the query interface <code>ucp_worker_query</code>.</td>
</tr>
<tr>
<td>ucs_cpu_set_t</td>
<td>cpu_mask</td>
<td>Mask of which CPUs worker resources should preferably be allocated on. This value is optional. If it's not set (along with its corresponding bit in the field_mask - UCP_WORKER_PARAM_FIELD_CPU_MASK), resources are allocated according to system's default policy.</td>
</tr>
<tr>
<td>unsigned</td>
<td>events</td>
<td>Mask of events (ucp_wakeup_event_t) which are expected on wakeup. This value is optional. If it's not set (along with its corresponding bit in the field_mask - UCP_WORKER_PARAM_FIELD_EVENTS), all types of events will trigger on wakeup.</td>
</tr>
<tr>
<td>void *</td>
<td>user_data</td>
<td>User data associated with the current worker. This value is optional. If it's not set (along with its corresponding bit in the field_mask - UCP_WORKER_PARAM_FIELD_USER_DATA), it will default to NULL.</td>
</tr>
</tbody>
</table>
| int           | event_fd       | External event file descriptor. This value is optional. If UCP_WORKER_PARAM_FIELD_EVENT_FD is set in the field_mask, events on the worker will be reported on the provided event file descriptor. In this case, calling `ucp_worker_get_efd` will result in an error. The provided file descriptor must be capable of aggregating notifications for arbitrary events, for example `epoll(7)` on Linux systems. `user_data` will be used as the event user-data on systems which support it. For example, on Linux, it will be placed in `epoll_data_t::ptr`, when returned from `epoll_wait(2)`. Otherwise, events will be reported to the event file descriptor returned from `ucp_worker_get_efd()`.

© 2021 Unified Communication X (UCX). All rights reserved.
6.3.2.3 struct ucp_listener_attr

The structure defines the attributes which characterize the particular listener.

Examples

ucp_client_server.c.

Data Fields

<table>
<thead>
<tr>
<th>data type</th>
<th>field</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint64_t</td>
<td>field_mask</td>
<td>Mask of valid fields in this structure, using bits from ucp_listener_attr_field. Fields not specified in this mask will be ignored. Provides ABI compatibility with respect to adding new fields.</td>
</tr>
<tr>
<td>struct sockaddr_storage</td>
<td>sockaddr</td>
<td>Sockaddr on which this listener is listening for incoming connection requests.</td>
</tr>
</tbody>
</table>

6.3.2.4 struct ucp_conn_request_attr

The structure defines the attributes that characterize the particular connection request received on the server side.

Examples

ucp_client_server.c.

Data Fields

<table>
<thead>
<tr>
<th>data type</th>
<th>field</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint64_t</td>
<td>field_mask</td>
<td>Mask of valid fields in this structure, using bits from ucp_conn_request_attr_field. Fields not specified in this mask will be ignored. Provides ABI compatibility with respect to adding new fields.</td>
</tr>
<tr>
<td>struct sockaddr_storage</td>
<td>client_address</td>
<td>The address of the remote client that sent the connection request to the server.</td>
</tr>
</tbody>
</table>

6.3.2.5 struct ucp_listener_params

This structure defines parameters for ucp_listener_create, which is used to listen for incoming client/server connections.

Examples

ucp_client_server.c.

Data Fields

<table>
<thead>
<tr>
<th>data type</th>
<th>field</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint64_t</td>
<td>field_mask</td>
<td>Mask of valid fields in this structure, using bits from ucp_listener_params_field. Fields not specified in this mask will be ignored. Provides ABI compatibility with respect to adding new fields.</td>
</tr>
<tr>
<td>ucs_sock_addr_t</td>
<td>sockaddr</td>
<td>An address in the form of a sockaddr. This field is mandatory for filling (along with its corresponding bit in the field_mask - UCP_LISTENER_PARAM_FIELD_SOCK_ADDR). The ucp_listener_create routine will return with an error if sockaddr is not specified.</td>
</tr>
</tbody>
</table>
6.3 UCP Worker

Data Fields

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ucp_listener_accept_handler_t</td>
<td>accept_handler</td>
<td>Handler to endpoint creation in a client-server connection flow. In order for the callback inside this handler to be invoked, the UCP_LISTENER_PARAM_FIELD_ACCEPT_HANDLER needs to be set in the field_mask.</td>
</tr>
<tr>
<td>ucp_listener_conn_handler_t</td>
<td>conn_handler</td>
<td>Handler of an incoming connection request in a client-server connection flow. In order for the callback inside this handler to be invoked, the UCP_LISTENER_PARAM_FIELD_CONN_HANDLER needs to be set in the field_mask.</td>
</tr>
</tbody>
</table>

6.3.2.6 struct ucp_am_handler_param

Examples

ucp_client_server.c.

Data Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>field_mask</td>
<td>uint64_t</td>
<td>Mask of valid fields in this structure, using bits from ucp_am_handler_param_field. Fields not specified in this mask will be ignored. Provides ABI compatibility with respect to adding new fields.</td>
</tr>
<tr>
<td>id</td>
<td>unsigned</td>
<td>Active Message id.</td>
</tr>
<tr>
<td>flags</td>
<td>uint32_t</td>
<td>Handler flags as defined by ucp_am_cb_flags.</td>
</tr>
<tr>
<td>cb</td>
<td>ucp_am_recv_callback_t</td>
<td>Active Message callback. To clear the already set callback, this value should be set to NULL.</td>
</tr>
<tr>
<td>arg</td>
<td>void *</td>
<td>Active Message argument, which will be passed in to every invocation of ucp_am_recv_callback_t function as the arg argument.</td>
</tr>
</tbody>
</table>

6.3.2.7 struct ucp_am_recv_param

Examples

ucp_client_server.c.

Data Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>recv_attr</td>
<td>uint64_t</td>
<td>Mask of valid fields in this structure and receive operation flags, using bits from ucp_am_recv_attr_t. Fields not specified in this mask will be ignored. Provides ABI compatibility with respect to adding new fields.</td>
</tr>
<tr>
<td>reply_ep</td>
<td>ucp_ep_h</td>
<td>Endpoint, which can be used for reply to this message.</td>
</tr>
</tbody>
</table>

6.3.2.8 struct ucp_listener_accept_handler

Deprecated Replaced by ucp_listener_conn_handler_t.
Data Fields

<table>
<thead>
<tr>
<th>ucp_listener_accept_callback_t</th>
<th>cb</th>
<th>Endpoint creation callback</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>void * arg</td>
<td>User defined argument for the callback</td>
</tr>
</tbody>
</table>

6.3.2.9 struct ucp_listener_conn_handler

This structure is used for handling an incoming connection request on the listener. Setting this type of handler allows creating an endpoint on any other worker and not limited to the worker on which the listener was created.

Note

- Other than communication progress routines, it is allowed to call all other communication routines from the callback in the struct.
- The callback is thread safe with respect to the worker it is invoked on.
- It is the user's responsibility to avoid potential dead lock accessing different worker.

Data Fields

<table>
<thead>
<tr>
<th>ucp_listener_conn_callback_t</th>
<th>cb</th>
<th>Connection request callback</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>void * arg</td>
<td>User defined argument for the callback</td>
</tr>
</tbody>
</table>

6.3.3 Typedef Documentation

6.3.3.1 ucp_worker_attr_t

typedef struct ucp_worker_attr ucp_worker_attr_t

The structure defines the attributes which characterize the particular worker.

6.3.3.2 ucp_worker_params_t

typedef struct ucp_worker_params ucp_worker_params_t

The structure defines the parameters that are used for the UCP worker tuning during the UCP worker creation.

6.3.3.3 ucp_listener_attr_t

typedef struct ucp_listener_attr ucp_listener_attr_t

The structure defines the attributes which characterize the particular listener.

6.3.3.4 ucp_conn_request_attr_t

typedef struct ucp_conn_request_attr ucp_conn_request_attr_t

The structure defines the attributes that characterize the particular connection request received on the server side.
6.3.3.5 ucp_listener_params_t

typedef struct ucp_listener_params ucp_listener_params_t

This structure defines parameters for ucp_listener_create, which is used to listen for incoming client/server connections.

6.3.3.6 ucp_am_handler_param_t

typedef struct ucp_am_handler_param ucp_am_handler_param_t

6.3.3.7 ucp_listener_accept_handler_t

typedef struct ucp_listener_accept_handler ucp_listener_accept_handler_t

**Deprecated**  Replaced by ucp_listener_conn_handler_t.

6.3.3.8 ucp_am_recv_param_t

typedef struct ucp_am_recv_param ucp_am_recv_param_t

6.3.3.9 ucp_address_t

typedef struct ucp_address ucp_address_t

The address handle is an opaque object that is used as an identifier for a worker instance.

6.3.3.10 ucp_listener_h

typedef struct ucp_listener* ucp_listener_h

The listener handle is an opaque object that is used for listening on a specific address and accepting connections from clients.

6.3.3.11 ucp_worker_h

typedef struct ucp_worker* ucp_worker_h

UCP worker is an opaque object representing the communication context. The worker represents an instance of a local communication resource and the progress engine associated with it. The progress engine is a construct that is responsible for asynchronous and independent progress of communication directives. The progress engine could be implemented in hardware or software. The worker object abstracts an instance of network resources such as a host channel adapter port, network interface, or multiple resources such as multiple network interfaces or communication ports. It could also represent virtual communication resources that are defined across multiple devices. Although the worker can represent multiple network resources, it is associated with a single UCX application context. All communication functions require a context to perform the operation on the dedicated hardware resource(s) and an endpoint to address the destination.
Worker are parallel “threading points” that an upper layer may use to optimize concurrent communications.

### 6.3.3.12 ucp_listener_accept_callback_t

```c
typedef void(* ucp_listener_accept_callback_t) (ucp_ep_h ep, void *arg)
```

This callback routine is invoked on the server side upon creating a connection to a remote client. The user can pass an argument to this callback. The user is responsible for releasing the `ep` handle using the `ucp_ep_destroy()` routine.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ep</code></td>
<td>Handle to a newly created endpoint which is connected to the remote peer which has initiated the connection.</td>
</tr>
<tr>
<td><code>arg</code></td>
<td>User’s argument for the callback.</td>
</tr>
</tbody>
</table>

### 6.3.3.13 ucp_listener_conn_callback_t

```c
typedef void(* ucp_listener_conn_callback_t) (ucp_conn_request_h conn_request, void *arg)
```

This callback routine is invoked on the server side to handle incoming connections from remote clients. The user can pass an argument to this callback. The `conn_request` handle has to be released, either by `ucp_ep_create` or `ucp_listener_reject` routine.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>conn_request</code></td>
<td>Connection request handle.</td>
</tr>
<tr>
<td><code>arg</code></td>
<td>User’s argument for the callback.</td>
</tr>
</tbody>
</table>

### 6.3.3.14 ucp_listener_conn_handler_t

```c
typedef struct ucp_listener_conn_handler ucp_listener_conn_handler_t
```

This structure is used for handling an incoming connection request on the listener. Setting this type of handler allows creating an endpoint on any other worker and not limited to the worker on which the listener was created.

**Note**

- Other than communication progress routines, it is allowed to call all other communication routines from the callback in the struct.
- The callback is thread safe with respect to the worker it is invoked on.
- It is the user's responsibility to avoid potential deadlock accessing different worker.

### 6.3.3.15 ucp_wakeup_event_t

```c
typedef enum ucp_wakeup_event_types ucp_wakeup_event_t
```

© 2021 Unified Communication X (UCX). All rights reserved.
The enumeration allows specifying which events are expected on wakeup. Empty events are possible for any type of event except for UCP_WAKEUP_TX and UCP_WAKEUP_RX.

Note

Send completions are reported by POLLIN-like events (see poll man page). Since outgoing operations can be initiated at any time, UCP does not generate POLLOUT-like events, although it must be noted that outgoing operations may be queued depending upon resource availability.

6.3.4 Enumeration Type Documentation

6.3.4.1 ucp_worker_params_field

enum ucp_worker_params_field

The enumeration allows specifying which fields in ucp_worker_params_t are present. It is used to enable backward compatibility support.

**Enumerator**

<table>
<thead>
<tr>
<th>UCP_WORKER_PARAM_FIELD_THREAD_MODE</th>
<th>UCP thread mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCP_WORKER_PARAM_FIELD_CPU_MASK</td>
<td>Worker's CPU bitmap</td>
</tr>
<tr>
<td>UCP_WORKER_PARAM_FIELD_EVENTS</td>
<td>Worker's events bitmap</td>
</tr>
<tr>
<td>UCP_WORKER_PARAM_FIELD_USER_DATA</td>
<td>User data</td>
</tr>
<tr>
<td>UCP_WORKER_PARAM_FIELD_EVENT_FD</td>
<td>External event file descriptor</td>
</tr>
</tbody>
</table>

6.3.4.2 ucp_listener_params_field

enum ucp_listener_params_field

The enumeration allows specifying which fields in ucp_listener_params_t are present. It is used to enable backward compatibility support.

**Enumerator**

<table>
<thead>
<tr>
<th>UCP_LISTENER_PARAM_FIELD_SOCK_ADDR</th>
<th>Sock address and length.</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCP_LISTENER_PARAM_FIELD_ACCEPT_HAN--</td>
<td>User's callback and argument for handling the creation of an endpoint. User's callback and argument for handling the incoming connection request.</td>
</tr>
<tr>
<td>DLER</td>
<td></td>
</tr>
<tr>
<td>UCP_LISTENER_PARAM_FIELD_CONN_HANDL--</td>
<td></td>
</tr>
<tr>
<td>ER</td>
<td></td>
</tr>
</tbody>
</table>

6.3.4.3 ucp_worker_address_flags_t

enum ucp_worker_address_flags_t

The enumeration list describes possible UCP worker address flags, indicating what needs to be included to the worker address returned by ucp_worker_query() routine.

© 2021 Unified Communication X (UCX). All rights reserved.
6.3.4.4 ucp_worker_attr_field

enum ucp_worker_attr_field

The enumeration allows specifying which fields in ucp_worker_attr_t are present. It is used to enable backward compatibility support.

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCP_WORKER_ATTR_FIELD_THREAD_MODE</td>
<td>UCP thread mode</td>
</tr>
<tr>
<td>UCP_WORKER_ATTR_FIELD_ADDRESS</td>
<td>UCP address</td>
</tr>
<tr>
<td>UCP_WORKER_ATTR_FIELD_ADDRESS_FLAGS</td>
<td>UCP address flags</td>
</tr>
<tr>
<td>UCP_WORKER_ATTR_FIELD_MAX_AM_HEADER</td>
<td>Maximal header size used by UCP AM API</td>
</tr>
</tbody>
</table>

6.3.4.5 ucp_listener_attr_field

enum ucp_listener_attr_field

The enumeration allows specifying which fields in ucp_listener_attr_t are present. It is used to enable backward compatibility support.

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCP_LISTENER_ATTR_FIELD_SOCKADDR</td>
<td>Sockaddr used for listening</td>
</tr>
</tbody>
</table>

6.3.4.6 ucp_conn_request_attr_field

enum ucp_conn_request_attr_field

The enumeration allows specifying which fields in ucp_conn_request_attr_t are present. It is used to enable backward compatibility support.

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCP_CONN_REQUEST_ATTR_FIELD_CLIENT_ADDR</td>
<td>Client's address</td>
</tr>
</tbody>
</table>
6.3.4.7 ucp_am_cb_flags

enum ucp_am_cb_flags
Flags that indicate how to handle UCP Active Messages. Currently only UCP_AM_FLAG_WHOLE_MSG is supported, which indicates the entire message is handled in one callback.

Enumerator

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCP_AM_FLAG_WHOLE_MSG</td>
<td></td>
</tr>
</tbody>
</table>

6.3.4.8 ucp_send_am_flags

enum ucp_send_am_flags
Flags dictate the behavior of ucp_am_send_nb and ucp_am_send_nbx routines.

Enumerator

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCP_AM_SEND_FLAG_REPLY</td>
<td>Force relevant reply endpoint to be passed to the data callback on the receiver.</td>
</tr>
<tr>
<td>UCP_AM_SEND_FLAG_EAGER</td>
<td>Force UCP to use only eager protocol for AM sends.</td>
</tr>
<tr>
<td>UCP_AM_SEND_FLAG_RNDV</td>
<td>Force UCP to use only rendezvous protocol for AM sends.</td>
</tr>
<tr>
<td>UCP_AM_SEND_REPLY</td>
<td>Backward compatibility.</td>
</tr>
</tbody>
</table>

6.3.4.9 ucp_wakeup_event_types

enum ucp_wakeup_event_types
The enumeration allows specifying which events are expected on wakeup. Empty events are possible for any type of event except for UCP_WAKEUP_TX and UCP_WAKEUP_RX.

Note
Send completions are reported by POLLIN-like events (see poll man page). Since outgoing operations can be initiated at any time, UCP does not generate POLLOUT-like events, although it must be noted that outgoing operations may be queued depending upon resource availability.

Enumerator

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCP_WAKEUP_RMA</td>
<td>Remote memory access send completion</td>
</tr>
<tr>
<td>UCP_WAKEUP_AMO</td>
<td>Atomic operation send completion</td>
</tr>
<tr>
<td>UCP_WAKEUP_TAG_SEND</td>
<td>Tag send completion</td>
</tr>
<tr>
<td>UCP_WAKEUP_TAG_RECV</td>
<td>Tag receive completion</td>
</tr>
<tr>
<td>UCP_WAKEUP_TX</td>
<td>This event type will generate an event on completion of any outgoing operation (complete or partial, according to the underlying protocol) for any type of transfer (send, atomic, or RMA).</td>
</tr>
<tr>
<td>UCP_WAKEUP_RX</td>
<td>This event type will generate an event on completion of any receive operation (complete or partial, according to the underlying protocol).</td>
</tr>
<tr>
<td>UCP_WAKEUP_EDGE</td>
<td>Use edge-triggered wakeup. The event file descriptor will be signaled only for new events, rather than existing ones.</td>
</tr>
</tbody>
</table>
6.3.5 Function Documentation

6.3.5.1 ucp_worker_create()

```c
ucs_status_t ucp_worker_create (  
    ucp_context_h context,  
    const ucp_worker_params_t * params,  
    ucp_worker_h * worker_p )
```

This routine allocates and initializes a worker object. Each worker is associated with one and only one application context. In the same time, an application context can create multiple workers in order to enable concurrent access to communication resources. For example, application can allocate a dedicated worker for each application thread, where every worker can be progressed independently of others.

Note
The worker object is allocated within context of the calling thread

Parameters

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>context</td>
<td>Handle to UCP application context.</td>
</tr>
<tr>
<td>in</td>
<td>params</td>
<td>User defined ucp_worker_params_t configurations for the UCP worker.</td>
</tr>
<tr>
<td>out</td>
<td>worker_p</td>
<td>A pointer to the worker object allocated by the UCP library</td>
</tr>
</tbody>
</table>

Returns
Error code as defined by ucs_status_t

Examples
ucp_client_server.c, and ucp_hello_world.c.

6.3.5.2 ucp_worker_destroy()

```c
void ucp_worker_destroy (  
    ucp_worker_h worker )
```

This routine releases the resources associated with a UCP worker.

Warning
Once the UCP worker destroy the worker handle cannot be used with any UCP routine.

The destroy process releases and shuts down all resources associated with the worker.

Parameters

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>worker</td>
<td>Worker object to destroy.</td>
</tr>
</tbody>
</table>
Examples

ucp_client_server.c, and ucp_hello_world.c.

6.3.5.3 ucp_worker_query()

```c
ucs_status_t ucp_worker_query (  
    ucp_worker_h worker,  
    ucp_worker_attr_t *attr )
```

This routine fetches information about the worker.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>worker</th>
<th>Worker object to query.</th>
</tr>
</thead>
<tbody>
<tr>
<td>out</td>
<td>attr</td>
<td>Filled with attributes of worker.</td>
</tr>
</tbody>
</table>

Returns

Error code as defined by ucs_status_t

6.3.5.4 ucp_worker_print_info()

```c
void ucp_worker_print_info (  
    ucp_worker_h worker,  
    FILE *stream )
```

This routine prints information about the protocols being used, thresholds, UCT transport methods, and other useful information associated with the worker.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>worker</th>
<th>Worker object to print information for.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>stream</td>
<td>Output stream to print the information to.</td>
</tr>
</tbody>
</table>

6.3.5.5 ucp_worker_get_address()

```c
ucs_status_t ucp_worker_get_address (  
    ucp_worker_h worker,  
    ucp_address_t **address_p,  
    size_t *address_length_p )
```

This routine returns the address of the worker object. This address can be passed to remote instances of the UCP library in order to connect to this worker. The memory for the address handle is allocated by this function, and must be released by using ucp_worker_release_address() routine.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>worker</th>
<th>Worker object whose address to return.</th>
</tr>
</thead>
<tbody>
<tr>
<td>out</td>
<td>address_p</td>
<td>A pointer to the worker address.</td>
</tr>
</tbody>
</table>
Parameters

| out | address_length | The size in bytes of the address. |

Returns

Error code as defined by ucs_status_t

Examples

ucp_hello_world.c.

6.3.5.6 ucp_worker_release_address()

void ucp_worker_release_address (  
    ucp_worker_h worker,  
    ucp_address_t * address )

This routine release an address handle associated within the worker object.

Warning

Once the address released the address handle cannot be used with any UCP routine.

Parameters

| in | worker | Worker object that is associated with the address object. |
| in | address | Address to release; the address object has to be allocated using ucp_worker_get_address() routine. |

Examples

ucp_hello_world.c.

6.3.5.7 ucp_worker_progress()

unsigned ucp_worker_progress (  
    ucp_worker_h worker )

This routine explicitly progresses all communication operations on a worker.

Note

- Typically, request wait and test routines call this routine to progress any outstanding operations.
- Transport layers, implementing asynchronous progress using threads, require callbacks and other user code to be thread safe.
- The state of communication can be advanced (progressed) by blocking routines. Nevertheless, the non-blocking routines can not be used for communication progress.
6.3 UCP Worker

Parameters

| in  | worker | Worker to progress. |

Returns

Non-zero if any communication was progressed, zero otherwise.

Examples

ucp_client_server.c, and ucp_hello_world.c.

6.3.5.8 ucp_stream_worker_poll()

```
ssize_t ucp_stream_worker_poll (   
    ucp_worker_h worker,   
    ucp_stream_poll_ep_t * poll_eps,   
    size_t max_eps,   
    unsigned flags )
```

This non-blocking routine returns endpoints on a worker which are ready to consume streaming data. The ready endpoints are placed in `poll_eps` array, and the function return value indicates how many are there.

Parameters

| in  | worker | Worker to poll. |
| in  | max_eps | Maximal number of endpoints which should be filled in `poll_eps`. |
| out | poll_eps | Pointer to array of endpoints, should be allocated by user. |
| in  | flags | Reserved for future use. |

Returns

Negative value indicates an error according to `ucs_status_t`. On success, non-negative value (less or equal `max_eps`) indicates actual number of endpoints filled in `poll_eps` array.

6.3.5.9 ucp_listener_create()

```
ucs_status_t ucp_listener_create (   
    ucp_worker_h worker,   
    const ucp_listener_params_t * params,   
    ucp_listener_h * listener_p )
```

This routine binds the worker object to a `ucs_sock_addr_t` sockaddr which is set by the user. The worker will listen to incoming connection requests and upon receiving such a request from the remote peer, an endpoint to it will be created. The user's call-back will be invoked once the endpoint is created.

Parameters

| in  | worker | Worker object that is associated with the params object. |
| in  | params | User defined `ucp_listener_params_t` configurations for the `ucp_listener_h`. |
| out | listener_p | A handle to the created listener, can be released by calling `ucp_listener_destroy`. |
Returns

Error code as defined by `ucs_status_t`

Examples

`ucp_client_server.c`

---

### 6.3.5.10 ucp_listener_destroy()

```c
void ucp_listener_destroy (  
    ucp_listener_h listener  
)
```

This routine unbinds the worker from the given handle and stops listening for incoming connection requests on it.

Parameters

| in  | `listener` | A handle to the listener to stop listening on. |

Examples

`ucp_client_server.c`

---

### 6.3.5.11 ucp_listener_query()

```c
ucs_status_t ucp_listener_query (  
    ucp_listener_h listener,  
    ucp_listener_attr_t * attr  
)
```

This routine fetches information about the listener.

Parameters

| in   | `listener` | listener object to query. |
| out  | `attr`     | Filled with attributes of the listener. |

Returns

Error code as defined by `ucs_status_t`

Examples

`ucp_client_server.c`

---

### 6.3.5.12 ucp_conn_request_query()

```c
ucs_status_t ucp_conn_request_query (  
    ucp_conn_request_h conn_request,  
    ucp_conn_request_attr_t * attr  
)
```

This routine fetches information about the connection request.
6.3 UCP Worker

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>conn_request</th>
<th>connection request object to query.</th>
</tr>
</thead>
<tbody>
<tr>
<td>out</td>
<td>attr</td>
<td>Filled with attributes of the connection request.</td>
</tr>
</tbody>
</table>

Returns

Error code as defined by ucs_status_t

Examples

ucp_client_server.c.

6.3.5.13 ucp_listener_reject()

```c
ucs_status_t ucp_listener_reject (  
  ucp_listener_h listener,  
  ucp_conn_request_h conn_request )
```

Reject the incoming connection request and release associated resources. If the remote initiator endpoint has set an ucp_ep_params_t::err_handler, it will be invoked with status UCS_ERR_REJECTED.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>listener</th>
<th>Handle to the listener on which the connection request was received.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>conn_request</td>
<td>Handle to the connection request to reject.</td>
</tr>
</tbody>
</table>

Returns

Error code as defined by ucs_status_t

Examples

ucp_client_server.c.

6.3.5.14 ucp_worker_set_am_handler()

```c
ucs_status_t ucp_worker_set_am_handler (  
  ucp_worker_h worker,  
  uint16_t id,  
  ucp_am_callback_t cb,  
  void * arg,  
  uint32_t flags )
```

This routine installs a user defined callback to handle incoming Active Messages with a specific id. This callback is called whenever an Active Message that was sent from the remote peer by ucp_am_send_nb is received on this worker.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>worker</th>
<th>UCP worker on which to set the Active Message handler.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>id</td>
<td>Active Message id.</td>
</tr>
</tbody>
</table>
### ucp_worker_set_am_recv_handler()

```c
ucs_status_t ucp_worker_set_am_recv_handler (ucp_worker_h worker, const ucp_am_handler_param_t *param)
```

This routine installs a user defined callback to handle incoming Active Messages with a specific id. This callback is called whenever an Active Message that was sent from the remote peer by `ucp_am_send_nb` is received on this worker.

**Warning**

Handlers set by this function are not compatible with `ucp_am_send_nb` routine.

#### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>worker</code></td>
<td>UCP worker on which to set the Active Message handler.</td>
</tr>
<tr>
<td><code>param</code></td>
<td>Active Message handler parameters, as defined by <code>ucp_am_handler_param_t</code>.</td>
</tr>
</tbody>
</table>

**Returns**

- error code if the worker does not support Active Messages or requested callback flags.

#### Examples

- `ucp_client_server.c`

### ucp_worker_fence()

```c
ucs_status_t ucp_worker_fence (ucp_worker_h worker)
```

This routine ensures ordering of non-blocking communication operations on the UCP worker. Communication operations issued on the `worker` prior to this call are guaranteed to be completed before any subsequent communication operations to the same `worker` which follow the call to `fence`.

#### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>worker</code></td>
<td>UCP worker which is being fenced.</td>
</tr>
</tbody>
</table>

**Returns**

- error code if the worker does not support Active Messages or requested callback flags.

### Module Documentation

#### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cb</code></td>
<td>Active Message callback. NULL to clear.</td>
</tr>
<tr>
<td><code>arg</code></td>
<td>Active Message argument, which will be passed in to every invocation of the callback as the arg argument.</td>
</tr>
<tr>
<td><code>flags</code></td>
<td>Dictates how an Active Message is handled on the remote endpoint. Currently only UCP_AM_FLAG_WHOLE_MSG is supported, which indicates the callback will not be invoked until all data has arrived.</td>
</tr>
</tbody>
</table>

**Returns**

- error code if the worker does not support Active Messages or requested callback flags.
6.3 UCP Worker

Note

The primary difference between `ucp_worker_fence()` and the `ucp_worker_flush_nb()` is the fact the fence routine does not guarantee completion of the operations on the call return but only ensures the order between communication operations. The `flush` operation on return guarantees that all operations are completed and corresponding memory regions were updated.

Parameters

| in | worker | UCP worker |
|-----------------------------|

Returns

Error code as defined by `ucs_status_t`

6.3.5.17 ucp_worker_flush_nb()

```c
ucs_status_ptr_t ucp_worker_flush_nb (
    ucp_worker_h worker,
    unsigned flags,
    ucp_send_callback_t cb )
```

This routine flushes all outstanding AMO and RMA communications on the `worker`. All the AMO and RMA operations issued on the `worker` prior to this call are completed both at the origin and at the target when this call returns.

Note

For description of the differences between `flush` and `fence` operations please see `ucp_worker_fence()`

Parameters

| in | worker | UCP worker |
|-----------------------------|
| in | flags | Flags for flush operation. Reserved for future use. |
| in | cb | Callback which will be called when the flush operation completes. |

Returns

- `NULL` - The flush operation was completed immediately.
- `UCS_PTR_IS_ERR(_ptr)` - The flush operation failed.
- otherwise - Flush operation was scheduled and can be completed in any point in time. The request handle is returned to the application in order to track progress. The application is responsible for releasing the handle using `ucp_request_free()` routine.

6.3.5.18 ucp_worker_flush_nbx()

```c
ucs_status_ptr_t ucp_worker_flush_nbx ( 
    ucp_worker_h worker, 
    const ucp_request_param_t *param )
```

This routine flushes all outstanding AMO and RMA communications on the `worker`. All the AMO and RMA operations issued on the `worker` prior to this call are completed both at the origin and at the target when this call returns.
Note

For description of the differences between flush and fence operations please see ucp_worker_fence()

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>worker</th>
<th>UCP worker.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>param</td>
<td>Operation parameters, see ucp_request_param_t</td>
</tr>
</tbody>
</table>

Returns

NULL - The flush operation was completed immediately.
UCS_PTR_IS_ERR(_ptr) - The flush operation failed.
otherwise - Flush operation was scheduled and can be completed in any point in time. The request handle is returned to the application in order to track progress.

6.3.5.19 ucp_worker_flush()

```c
ucs_status_t ucp_worker_flush (ucp_worker_h worker)
```

Deprecated Replaced by ucp_worker_flush_nb. The following example implements the same functionality using ucp_worker_flush_nb:

```c
ucs_status_t worker_flush(ucp_worker_h worker) {
    void *request = ucp_worker_flush_nb(worker);
    if (request == NULL) {
        return UCS_OK;
    } else if (UCS_PTR_IS_ERR(request)) {
        return UCS_PTR_STATUS(request);
    } else {
        ucs_status_t status;
        do {
            ucp_worker_progress(worker);
            status = ucp_request_check_status(request);
        } while (status == UCS_INPROGRESS);
        ucp_request_release(request);
        return status;
    }
}
```

This routine flushes all outstanding AMO and RMA communications on the worker. All the AMO and RMA operations issued on the worker prior to this call are completed both at the origin and at the target when this call returns.

Note

For description of the differences between flush and fence operations please see ucp_worker_fence()

Parameters

| in | worker | UCP worker. |

Returns

Error code as defined by ucs_status_t
6.4 UCP Memory routines

Data Structures

- struct ucp_mem_map_params
  
  Tuning parameters for the UCP memory mapping. More...

- struct ucp_mem_advise_params
  
  Tuning parameters for the UCP memory advice. More...

- struct ucp_mem_attr
  
  Attributes of the UCP Memory handle, filled by ucp_mem_query function. More...

Typedefs

- typedef struct ucp_mem_map_params ucp_mem_map_params_t
  
  Tuning parameters for the UCP memory mapping.

- typedef enum ucp_mem_advice ucp_mem_advice_t
  
  list of UCP memory use advice.

- typedef struct ucp_mem_advise_params ucp_mem_advise_params_t
  
  Tuning parameters for the UCP memory advice.

- typedef struct ucp_rkey * ucp_rkey_h
  
  UCP Remote memory handle.

- typedef struct ucp_mem * ucp_mem_h
  
  UCP Memory handle.

- typedef struct ucp_mem_attr ucp_mem_attr_t
  
  Attributes of the UCP Memory handle, filled by ucp_mem_query function.

Enumerations

- enum ucp_mem_map_params_field {
    UCP_MEM_MAP_PARAM_FIELD_ADDRESS = UCS_BIT(0),
    UCP_MEM_MAP_PARAM_FIELD_LENGTH = UCS_BIT(1),
    UCP_MEM_MAP_PARAM_FIELD_FLAGS = UCS_BIT(2),
    UCP_MEM_MAP_PARAM_FIELD_PROT = UCS_BIT(3),
    UCP_MEM_MAP_PARAM_FIELD_MEMORY_TYPE = UCS_BIT(4)
} 
  
  UCP memory mapping parameters field mask.

- enum ucp_mem_advise_params_field {
    UCP_MEM_ADVISE_PARAM_FIELD_ADDRESS = UCS_BIT(0),
    UCP_MEM_ADVISE_PARAM_FIELD_LENGTH = UCS_BIT(1),
    UCP_MEM_ADVISE_PARAM_FIELD_ADVICE
} 
  
  UCP memory advice parameters field mask.

- enum {
    UCP_MEM_MAP_NONBLOCK = UCS_BIT(0),
    UCP_MEM_MAP_ALLOCATE = UCS_BIT(1),
    UCP_MEM_MAP_FIXED
} 
  
  UCP memory mapping flags.

- enum {
    UCP_MEM_MAP_PROT_LOCAL_READ = UCS_BIT(0),
    UCP_MEM_MAP_PROT_LOCAL_WRITE
} 
  
  UCP memory mapping protection mode.

- enum ucp_mem_advice {
    UCP_MADV_NORMAL = 0,
    UCP_MADV_WILLNEED
} 
  
  list of UCP memory use advice.

- enum ucp_mem_attr_field {
    UCP_MEM_ATTR_FIELD_ADDRESS = UCS_BIT(0),
    UCP_MEM_ATTR_FIELD_LENGTH
} 
  
  UCP Memory handle attributes field mask.
Functions

- `ucs_status_t ucp_mem_map (ucp_context_h context, const ucp_mem_map_params_t *params, ucp_mem_h *memh_p)`
  
  Map or allocate memory for zero-copy operations.

- `ucs_status_t ucp_mem_unmap (ucp_context_h context, ucp_mem_h memh)`
  
  Unmap memory segment.

- `ucs_status_t ucp_mem_query (const ucp_mem_h memh, ucp_mem_attr_t *attr)`
  
  Query mapped memory segment.

- `void ucp_mem_print_info (const char *mem_size, ucp_context_h context, FILE *stream)`
  
  Print memory mapping information.

- `ucs_status_t ucp_mem_advise (ucp_context_h context, ucp_mem_h memh, ucp_mem_advise_params_t *params)`
  
  Give advice about the use of memory.

- `ucs_status_t ucp_rkey_pack (ucp_context_h context, ucp_mem_h memh, void **rkey_buffer_p, size_t *size_p)`
  
  Pack memory region remote access key.

- `void ucp_rkey_buffer_release (void *rkey_buffer)`
  
  Release packed remote key buffer.

- `ucs_status_t ucp_ep_rkey_unpack (ucp_ep_h ep, const void *rkey_buffer, ucp_rkey_h *rkey_p)`
  
  Create remote access key from packed buffer.

- `ucs_status_t ucp_rkey_ptr (ucp_rkey_h rkey, uint64_t raddr, void **addr_p)`
  
  Get a local pointer to remote memory.

- `void ucp_rkey_destroy (ucp_rkey_h rkey)`
  
  Destroy the remote key.

6.4.1 Detailed Description

UCP Memory routines

6.4.2 Data Structure Documentation

6.4.2.1 struct ucp_mem_map_params

The structure defines the parameters that are used for the UCP memory mapping tuning during the `ucp_mem_map` routine.

Data Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>field_mask</code></td>
<td>Mask of valid fields in this structure, using bits from ucp_mem_map_params_field. Fields not specified in this mask will be ignored. Provides ABI compatibility with respect to adding new fields.</td>
</tr>
<tr>
<td><code>address</code></td>
<td>If the address is not NULL, the routine maps (registers) the memory segment pointed to by this address. If the pointer is NULL, the library allocates mapped (registered) memory segment and returns its address in this argument. Therefore, this value is optional. If it's not set (along with its corresponding bit in the field_mask - UCP_MEM_MAP_PARAM_FIELD_ADDRESS), the ucp_mem_map routine will consider address as set to NULL and will allocate memory.</td>
</tr>
<tr>
<td><code>length</code></td>
<td>Length (in bytes) to allocate or map (register). This field is mandatory for filling (along with its corresponding bit in the field_mask - UCP_MEM_MAP_PARAM_FIELD_LENGTH). The ucp_mem_map routine will return with an error if the length isn't specified.</td>
</tr>
</tbody>
</table>
6.4 UCP Memory routines

Data Fields

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned int</td>
<td>flags</td>
<td>Allocation flags, e.g. UCP_MEM_MAP_NONBLOCK. This value is optional. If it's not set (along with its corresponding bit in the field_mask - UCP_MEM_MAP_PARAM_FIELD_FLAGS), the ucp_mem_map routine will consider the flags as set to zero.</td>
</tr>
<tr>
<td>unsigned int</td>
<td>prot</td>
<td>Memory protection mode, e.g. UCP_MEM_MAP_PROT_LOCAL_READ. This value is optional. If it's not set, the ucp_mem_map routine will consider the flags as set to UCP_MEM_MAP_PROT_LOCAL_READ</td>
</tr>
<tr>
<td>ucs_memory_type_t</td>
<td>memory_type</td>
<td></td>
</tr>
</tbody>
</table>

6.4.2.2 struct ucp_mem_advise_params

This structure defines the parameters that are used for the UCP memory advice tuning during the ucp_mem_advise routine.

Data Fields

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint64_t</td>
<td>field_mask</td>
<td>Mask of valid fields in this structure, using bits from ucp_mem_advise_params_field. All fields are mandatory. Provides ABI compatibility with respect to adding new fields.</td>
</tr>
<tr>
<td>void *</td>
<td>address</td>
<td>Memory base address.</td>
</tr>
<tr>
<td>size_t</td>
<td>length</td>
<td>Length (in bytes) to allocate or map (register).</td>
</tr>
<tr>
<td>ucp_mem_advice_t</td>
<td>advice</td>
<td>Memory use advice ucp_mem_advice</td>
</tr>
</tbody>
</table>

6.4.2.3 struct ucp_mem_attr

Data Fields

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint64_t</td>
<td>field_mask</td>
<td>Mask of valid fields in this structure, using bits from ucp_mem_attr_field. Fields not specified in this mask will be ignored. Provides ABI compatibility with respect to adding new fields.</td>
</tr>
<tr>
<td>void *</td>
<td>address</td>
<td>Address of the memory segment.</td>
</tr>
<tr>
<td>size_t</td>
<td>length</td>
<td>Size of the memory segment.</td>
</tr>
</tbody>
</table>

6.4.3 Typedef Documentation

6.4.3.1 ucp_mem_map_params_t

typedef struct ucp_mem_map_params ucp_mem_map_params_t

The structure defines the parameters that are used for the UCP memory mapping tuning during the ucp_mem_map routine.
6.4.3.2 ucp_mem_advice_t

typedef enum ucp_mem_advice ucp_mem_advice_t

The enumeration list describes memory advice supported by ucp_mem_advice() function.

6.4.3.3 ucp_mem_advise_params_t

typedef struct ucp_mem_advise_params ucp_mem_advise_params_t

This structure defines the parameters that are used for the UCP memory advice tuning during the ucp_mem_advise routine.

6.4.3.4 ucp_rkey_h

typedef struct ucp_rkey* ucp_rkey_h

Remote memory handle is an opaque object representing remote memory access information. Typically, the handle includes a memory access key and other network hardware specific information, which are input to remote memory access operations, such as PUT, GET, and ATOMIC. The object is communicated to remote peers to enable an access to the memory region.

6.4.3.5 ucp_mem_h

typedef struct ucp_mem* ucp_mem_h

Memory handle is an opaque object representing a memory region allocated through UCP library, which is optimized for remote memory access operations (zero-copy operations). The memory handle is a self-contained object, which includes the information required to access the memory region locally, while remote key is used to access it remotely. The memory could be registered to one or multiple network resources that are supported by UCP, such as InfiniBand, Gemini, and others.

6.4.3.6 ucp_mem_attr_t

typedef struct ucp_mem_attr ucp_mem_attr_t

6.4.4 Enumeration Type Documentation

6.4.4.1 ucp_mem_map_params_field

enum ucp_mem_map_params_field

The enumeration allows specifying which fields in ucp_mem_map_params_t are present. It is used to enable backward compatibility support.

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCP_MEM_MAP_PARAM_FIELD_ADDRESS</td>
<td>Address of the memory that will be used in the ucp_mem_map routine.</td>
</tr>
<tr>
<td>UCP_MEM_MAP_PARAM_FIELD_LENGTH</td>
<td>The size of memory that will be allocated or registered in the ucp_mem_map routine.</td>
</tr>
<tr>
<td>UCP_MEM_MAP_PARAM_FIELD_FLAGS</td>
<td>Allocation flags.</td>
</tr>
<tr>
<td>UCP_MEM_MAP_PARAM_FIELD_PROT</td>
<td>Memory protection mode.</td>
</tr>
<tr>
<td>UCP_MEM_MAP_PARAM_FIELD_MEMORY_TYPE</td>
<td>Memory type.</td>
</tr>
</tbody>
</table>
6.4.4.2 ucp_mem_advise_params_field

enum ucp_mem_advise_params_field

The enumeration allows specifying which fields in ucp_mem_advise_params_t are present. It is used to enable backward compatibility support.

**Enumerator**

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCP_MEM_ADVISE_PARAM_FIELD_ADDRESS</td>
<td>Address of the memory</td>
</tr>
<tr>
<td>UCP_MEM_ADVISE_PARAM_FIELD_LENGTH</td>
<td>The size of memory</td>
</tr>
<tr>
<td>UCP_MEM_ADVISE_PARAM_FIELD_ADVICE</td>
<td>Advice on memory usage</td>
</tr>
</tbody>
</table>

6.4.4.3 anonymous enum

**anonymous enum**

The enumeration list describes the memory mapping flags supported by ucp_mem_map() function.

**Enumerator**

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCP_MEM_MAP_NONBLOCK</td>
<td>Complete the mapping faster, possibly by not populating the pages in the mapping up-front, and mapping them later when they are accessed by communication routines.</td>
</tr>
<tr>
<td>UCP_MEM_MAP_ALLOCATE</td>
<td>Identify requirement for allocation, if passed address is not a null-pointer then it will be used as a hint or direct address for allocation.</td>
</tr>
<tr>
<td>UCP_MEM_MAP_FIXED</td>
<td>Don't interpret address as a hint: place the mapping at exactly that address. The address must be a multiple of the page size.</td>
</tr>
</tbody>
</table>

6.4.4.4 anonymous enum

**anonymous enum**

The enumeration list describes the memory mapping protections supported by the ucp_mem_map() function.

**Enumerator**

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCP_MEM_MAP_PROT_LOCAL_READ</td>
<td>Enable local read access.</td>
</tr>
<tr>
<td>UCP_MEM_MAP_PROT_LOCAL_WRITE</td>
<td>Enable local write access.</td>
</tr>
<tr>
<td>UCP_MEM_MAP_PROT_REMOTE_READ</td>
<td>Enable remote read access.</td>
</tr>
<tr>
<td>UCP_MEM_MAP_PROT_REMOTE_WRITE</td>
<td>Enable remote write access.</td>
</tr>
</tbody>
</table>

6.4.4.5 ucp_mem_advice

enum ucp_mem_advice
The enumeration list describes memory advice supported by `ucp_mem_advice()` function.

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCP_MADV_NORMAL</td>
<td>No special treatment</td>
</tr>
<tr>
<td>UCP_MADV_WILLNEED</td>
<td>can be used on the memory mapped with UCP_MEM_MAP_NONBLOCK to speed up memory mapping and to avoid page faults when the memory is accessed for the first time.</td>
</tr>
</tbody>
</table>

**6.4.6 ucp_mem_attr_field**

`enum ucp_mem_attr_field`

The enumeration allows specifying which fields in `ucp_mem_attr_t` are present. It is used to enable backward compatibility support.

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCP_MEM_ATTR_FIELD_ADDRESS</td>
<td>Virtual address</td>
</tr>
<tr>
<td>UCP_MEM_ATTR_FIELD_LENGTH</td>
<td>The size of memory region</td>
</tr>
</tbody>
</table>

**6.4.5 Function Documentation**

**6.4.5.1 ucp_mem_map()**

```c
ucs_status_t ucp_mem_map (  
    ucp_context_h context,  
    const ucp_mem_map_params_t * params,  
    ucp_mem_h * memh_p )
```

This routine maps or/and allocates a user-specified memory segment with UCP application context and the network resources associated with it. If the application specifies NULL as an address for the memory segment, the routine allocates a mapped memory segment and returns its address in the `address_p` argument. The network stack associated with an application context can typically send and receive data from the mapped memory without CPU intervention; some devices and associated network stacks require the memory to be mapped to send and receive data. The `memory handle` includes all information required to access the memory locally using UCP routines, while `remote registration handle` provides an information that is necessary for remote memory access.

**Note**

Another well-known terminology for the "map" operation that is typically used in the context of networking is memory "registration" or "pinning". The UCP library registers the memory the available hardware so it can be assessed directly by the hardware.

**Memory mapping assumptions:**

- A given memory segment can be mapped by several different communication stacks, if these are compatible.
- The `memh_p` handle returned may be used with any sub-region of the mapped memory.
- If a large segment is registered, and then segmented for subsequent use by a user, then the user is responsible for segmentation and subsequent management.
### 6.4 UCP Memory routines

Table 6.64: Matrix of behavior

<table>
<thead>
<tr>
<th>parameter/flag</th>
<th>NONBLOCK</th>
<th>ALLOCATE</th>
<th>FIXED</th>
<th>address</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>0/1 - the value only affects the register/mapping phase</td>
<td>0/1 - the value only affects the register/mapping phase</td>
<td>0/1 - the value only affects the register/mapping phase</td>
<td>0/1 - the value only affects the register/mapping phase</td>
<td>0/1 - the value only affects the register/mapping phase</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>error if length &gt; 0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>alloc+register</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>defined</td>
<td>error</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>error</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>defined</td>
<td>alloc+register, hint</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>defined</td>
<td>error</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>defined</td>
<td>alloc+register,fixed</td>
</tr>
</tbody>
</table>

**Note**

- **register** means that the memory will be registered in corresponding transports for RMA/AMO operations. This case intends that the memory was allocated by user before.
- **alloc+register** means that the memory will be allocated in the memory provided by the system and registered in corresponding transports for RMA/AMO operations.
- **alloc+register, hint** means that the memory will be allocated with using `ucp_mem_map_params::address` as a hint and registered in corresponding transports for RMA/AMO operations.
- **alloc+register, fixed** means that the memory will be allocated and registered in corresponding transports for RMA/AMO operations.
- **error** is an erroneous combination of the parameters.

**Parameters**

<table>
<thead>
<tr>
<th>in</th>
<th>context</th>
<th>Application context to map (register) and allocate the memory on.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>params</td>
<td>User defined <code>ucp_mem_map_params_t</code> configurations for the UCP memory handle.</td>
</tr>
<tr>
<td>out</td>
<td>memh→_p</td>
<td>UCP handle for the allocated segment.</td>
</tr>
</tbody>
</table>

**Returns**

Error code as defined by `ucs_status_t`

#### 6.4.5.2 ucp_mem_unmap()

```c
ucs_status_t ucp_mem_unmap (  
    ucp_context_h context,  
    ucp_mem_h memh  )
```

This routine unmaps a user specified memory segment, that was previously mapped using the `ucp_mem_map()` routine. The unmap routine will also release the resources associated with the memory handle. When the function returns, the `ucp_mem_h` and associated remote key will be invalid and cannot be used with any UCP routine.

**Note**

Another well known terminology for the "unmap" operation that is typically used in the context of networking is memory "de-registration". The UCP library de-registers the memory the available hardware so it can be returned back to the operation system.

Error cases:
• Once memory is unmapped a network access to the region may cause a failure.

Parameters

| in | context | Application context which was used to allocate/map the memory. |
| in | memh    | Handle to memory region. |

Returns

Error code as defined by ucs_status_t

6.4.5.3 ucp_mem_query()

```c
ucs_status_t ucp_mem_query (const ucp_mem_h memh, ucp_mem_attr_t *attr)
```

This routine returns address and length of memory segment mapped with ucp_mem_map() routine.

Parameters

| in | memh | Handle to memory region. |
| out| attr | Filled with attributes of the UCP memory handle. |

Returns

Error code as defined by ucs_status_t

6.4.5.4 ucp_mem_print_info()

```c
void ucp_mem_print_info (const char *mem_size, ucp_context_h context, FILE *stream)
```

This routine maps memory and prints information about the created memory handle: including the mapped memory length, the allocation method, and other useful information associated with the memory handle.

Parameters

| in | mem_size | Size of the memory to map. |
| in | context  | The context on which the memory is mapped. |
| in | stream   | Output stream on which to print the information. |

6.4.5.5 ucp_mem_advise()

```c
ucs_status_t ucp_mem_advise (c
```
This routine advises the UCP about how to handle memory range beginning at address and size of length bytes. This call does not influence the semantics of the application, but may influence its performance. The UCP may ignore the advice.

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>context</code></td>
<td>Application context which was used to allocate/map the memory.</td>
</tr>
<tr>
<td><code>memh</code></td>
<td>Handle to memory region.</td>
</tr>
<tr>
<td><code>params</code></td>
<td>Memory base address and length. The advice field is used to pass memory use advice as defined in the <code>ucp_mem_advice</code> list. The memory range must belong to the <code>memh</code>.</td>
</tr>
</tbody>
</table>

### Returns

Error code as defined by `ucs_status_t`.

---

**6.4.5.6 ucp_rkey_pack()**

```c
ucs_status_t ucp_rkey_pack(
    ucp_context_h context,
    ucp_mem_h memh,
    void **rkey_buffer_p,
    size_t *size_p
)
```

This routine allocates memory buffer and packs into the buffer a remote access key (RKEY) object. RKEY is an opaque object that provides the information that is necessary for remote memory access. This routine packs the RKEY object in a portable format such that the object can be unpacked on any platform supported by the UCP library. In order to release the memory buffer allocated by this routine the application is responsible for calling the `ucp_rkey_buffer_release()` routine.

### Note

- RKEYs for InfiniBand and Cray Aries networks typically includes InfiniBand and Aries key.
- In order to enable remote direct memory access to the memory associated with the memory handle the application is responsible for sharing the RKEY with the peers that will initiate the access.

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>context</code></td>
<td>Application context which was used to allocate/map the memory.</td>
</tr>
<tr>
<td><code>memh</code></td>
<td>Handle to memory region.</td>
</tr>
<tr>
<td><code>rkey_buffer_p</code></td>
<td>Memory buffer allocated by the library. The buffer contains packed RKEY.</td>
</tr>
<tr>
<td><code>size_p</code></td>
<td>Size (in bytes) of the packed RKEY.</td>
</tr>
</tbody>
</table>

### Returns

Error code as defined by `ucs_status_t`. 

---

© 2021 Unified Communication X (UCX). All rights reserved.
6.4.5.7 ucp_rkey_buffer_release()

```c
void ucp_rkey_buffer_release (  
    void * rkey_buffer  )
```

This routine releases the buffer that was allocated using `ucp_rkey_pack()`.

**Warning**

- Once memory is released an access to the memory may cause a failure.
- If the input memory address was not allocated using `ucp_rkey_pack()` routine the behaviour of this routine is undefined.

**Parameters**

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>In</td>
<td>rkey_buffer</td>
<td>Buffer to release.</td>
</tr>
</tbody>
</table>

6.4.5.8 ucp_ep_rkey_unpack()

```c
ucs_status_t ucp_ep_rkey_unpack (  
    ucp_ep_h ep,  
    const void * rkey_buffer,  
    ucp_rkey_h * rkey_p  )
```

This routine unpacks the remote key (RKEY) object into the local memory such that it can be accessed and used by UCP routines. The RKEY object has to be packed using the `ucp_rkey_pack()` routine. Application code should not make any changes to the content of the RKEY buffer.

**Note**

The application is responsible for releasing the RKEY object when it is no longer needed, by calling the `ucp_rkey_destroy()` routine.

The remote key object can be used for communications only on the endpoint on which it was unpacked.

**Parameters**

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>In</td>
<td>ep</td>
<td>Endpoint to access using the remote key.</td>
</tr>
<tr>
<td>In</td>
<td>rkey_buffer</td>
<td>Packed rkey.</td>
</tr>
<tr>
<td>Out</td>
<td>rkey_p</td>
<td>Remote key handle.</td>
</tr>
</tbody>
</table>

**Returns**

Error code as defined by `ucs_status_t`

6.4.5.9 ucp_rkey_ptr()

```c
ucs_status_t ucp_rkey_ptr (  
    ucp_rkey_h rkey,  
    uint64_t raddr,  
    void ** addr_p  )
```

This routine returns a local pointer to the remote memory described by the rkey.
6.4 UCP Memory routines

Note

This routine can return a valid pointer only for the endpoints that are reachable via shared memory.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>rkey</th>
<th>A remote key handle.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>raddr</td>
<td>A remote memory address within the memory area described by the rkey.</td>
</tr>
<tr>
<td>out</td>
<td>addr→_p</td>
<td>A pointer that can be used for direct access to the remote memory.</td>
</tr>
</tbody>
</table>

Returns

Error code as defined by ucs_status_t if the remote memory cannot be accessed directly or the remote memory address is not valid.

6.4.5.10 ucp_rkey_destroy()

void ucp_rkey_destroy(
    ucp_rkey_h rkey)

This routine destroys the RKEY object and the memory that was allocated using the ucp_ep_rkey_unpack() routine. This routine also releases any resources that are associated with the RKEY object.

Warning

- Once the RKEY object is released an access to the memory will cause an undefined failure.
- If the RKEY object was not created using ucp_ep_rkey_unpack() routine the behavior of this routine is undefined.
- The RKEY object must be destroyed after all outstanding operations which are using it are flushed, and before the endpoint on which it was unpacked is destroyed.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>rkey</th>
<th>Remote key to destroy.</th>
</tr>
</thead>
</table>
6.5 UCP Wake-up routines

Functions

- `ucs_status_t ucp_worker_get_efd (ucp_worker_h worker, int *fd)`
  Obtain an event file descriptor for event notification.
- `ucs_status_t ucp_worker_wait (ucp_worker_h worker)`
  Wait for an event of the worker.
- `void ucp_worker_wait_mem (ucp_worker_h worker, void *address)`
  Wait for memory update on the address.
- `ucs_status_t ucp_worker_arm (ucp_worker_h worker)`
  Turn on event notification for the next event.
- `ucs_status_t ucp_worker_signal (ucp_worker_h worker)`
  Cause an event of the worker.

6.5.1 Detailed Description

UCP Wake-up routines

6.5.2 Function Documentation

6.5.2.1 ucp_worker_get_efd()

```c
ucs_status_t ucp_worker_get_efd (  
  ucp_worker_h worker,  
  int *fd  
)
```

This routine returns a valid file descriptor for polling functions. The file descriptor will get signaled when an event occurs, as part of the wake-up mechanism. Signaling means a call to `poll()` or `select()` with this file descriptor will return at this point, with this descriptor marked as the reason (or one of the reasons) the function has returned. The user does not need to release the obtained file descriptor.

The wake-up mechanism exists to allow for the user process to register for notifications on events of the underlying interfaces, and wait until such occur. This is an alternative to repeated polling for request completion. The goal is to allow for waiting while consuming minimal resources from the system. This is recommended for cases where traffic is infrequent, and latency can be traded for lower resource consumption while waiting for it.

There are two alternative ways to use the wakeup mechanism: the first is the file descriptor obtained per worker (this function) and the second is the `ucp_worker_wait` function for waiting on the next event internally.

Note

UCP features have to be triggered with `UCP_FEATURE_WAKEUP` to select proper transport

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>worker</td>
<td>Worker of notified events.</td>
</tr>
<tr>
<td>out</td>
<td>fd</td>
<td>File descriptor.</td>
</tr>
</tbody>
</table>
6.5 UCP Wake-up routines

Returns

Error code as defined by ucs_status_t

Examples

ucp_hello_world.c.

6.5.2.2 ucp_worker_wait()

```c
ucs_status_t ucp_worker_wait (ucp_worker_h worker)
```

This routine waits (blocking) until an event has happened, as part of the wake-up mechanism.

This function is guaranteed to return only if new communication events occur on the worker. Therefore one must drain all existing events before waiting on the file descriptor. This can be achieved by calling ucp_worker_progress repeatedly until it returns 0.

There are two alternative ways to use the wake-up mechanism. The first is by polling on a per-worker file descriptor obtained from ucp_worker_get_efd. The second is by using this function to perform an internal wait for the next event associated with the specified worker.

Note

During the blocking call the wake-up mechanism relies on other means of notification and may not progress some of the requests as it would when calling ucp_worker_progress (which is not invoked in that duration). UCP features have to be triggered with UCP_FEATURE_WAKEUP to select proper transport

Parameters

| in    | worker | Worker to wait for events on. |

Returns

Error code as defined by ucs_status_t

Examples

ucp_hello_world.c.

6.5.2.3 ucp_worker_wait_mem()

```c
void ucp_worker_wait_mem (ucp_worker_h worker,
                         void * address)
```

This routine waits for a memory update at the local memory address. This is a blocking routine. The routine returns when the memory address is updated ("write") or an event occurs in the system.

This function is guaranteed to return only if new communication events occur on the worker or address is modified. Therefore one must drain all existing events before waiting on the file descriptor. This can be achieved by calling ucp_worker_progress repeatedly until it returns 0.
Note

This routine can be used by an application that executes busy-waiting loop checking for a memory update. Instead of continuous busy-waiting on an address the application can use `ucp_worker_wait_mem`, which may suspend execution until the memory is updated. The goal of the routine is to provide an opportunity for energy savings for architectures that support this functionality.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>worker</th>
<th>Worker to wait for updates on.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>address</td>
<td>Local memory address</td>
</tr>
</tbody>
</table>

6.5.2.4 ucp_worker_arm()

```c
ucs_status_t ucp_worker_arm (ucp_worker_h worker)
```

This routine needs to be called before waiting on each notification on this worker, so will typically be called once the processing of the previous event is over, as part of the wake-up mechanism.

The worker must be armed before waiting on an event (must be re-armed after it has been signaled for re-use) with `ucp_worker_arm`. The events triggering a signal of the file descriptor from `ucp_worker_get_efd` depend on the interfaces used by the worker and defined in the transport layer, and typically represent a request completion or newly available resources. It can also be triggered by calling `ucp_worker_signal`.

The file descriptor is guaranteed to become signaled only if new communication events occur on the worker. Therefore one must drain all existing events before waiting on the file descriptor. This can be achieved by calling `ucp_worker_progress` repeatedly until it returns 0.

```c
void application_initialization() {
  // should be called once in application init flow and before
  // process_comminucation() is used
  status = ucp_worker_get_efd(worker, &fd);
}
```

```c
void process_comminucation() {
  // should be called every time need to wait for some condition such as
  // ucp request completion in sleep mode.
  for (;;) {
    // check for stop condition as long as progress is made
    if (check_for_events()) {
      break;
    } else if (ucp_worker_progress(worker)) {
      // some progress happened but condition not met
      continue;
    }
    // arm the worker and clean-up fd
    status = ucp_worker_arm(worker);
    if (UCS_OK == status) {
      poll(&fds, nfds, timeout); // wait for events (sleep mode)
    } else if (UCS_ERR_BUSY == status) {
      continue; // could not arm, need to progress more
    } else {
      abort();
    }
  }
}
```

Note

UCP features have to be triggered with `UCP_FEATURE_WAKEUP` to select proper transport

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>worker</th>
<th>Worker of notified events.</th>
</tr>
</thead>
</table>
6.5 UCP Wake-up routines

Returns

UCS_OK The operation completed successfully. File descriptor will be signaled by new events.
UCS_ERR_BUSY There are unprocessed events which prevent the file descriptor from being armed. These
events should be removed by calling ucp_worker_progress(). The operation is not completed. File descriptor
will not be signaled by new events.
Other different error codes in case of issues.

Examples

ucp_hello_world.c.

6.5.2.5 ucp_worker_signal()

```c
ucp_status_t ucp_worker_signal (ucp_worker_h worker);
```

This routine signals that the event has happened, as part of the wake-up mechanism. This function causes a
blocking call to ucp_worker_wait or waiting on a file descriptor from ucp_worker_get_efd to return, even if no event
from the underlying interfaces has taken place.

Note

It’s safe to use this routine from any thread, even if UCX is compiled without multi-threading support and/or
initialized with any value of ucp_params_t::mt_workers_shared and ucp_worker_params_t::thread_mode pa-
rameters

Parameters

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>worker</td>
<td>Worker to wait for events on.</td>
</tr>
</tbody>
</table>

Returns

Error code as defined by ucs_status_t
6.6 UCP Endpoint

Data Structures

- `struct ucp_stream_poll_ep`
  
  Output parameter of `ucp_stream_worker_poll` function. More...

- `struct ucp_ep_params`
  
  Tuning parameters for the UCP endpoint. More...

Typedefs

- `typedef struct ucp_stream_poll_ep ucp_stream_poll_ep_t`
  
  Output parameter of `ucp_stream_worker_poll` function.

- `typedef struct ucp_ep * ucp_ep_h`
  
  UCP Endpoint.

- `typedef struct ucp_conn_request * ucp_conn_request_h`
  
  UCP connection request.

- `typedef ucs_status_t (ucp_am_callback_t) (void *arg, void *data, size_t length, ucp_ep_h reply_ep, unsigned flags)`
  
  Callback to process incoming Active Message.

- `typedef ucs_status_t (ucp_am_recv_callback_t) (void *arg, const void *header, size_t header_length, void *data, size_t length, const ucp_am_recv_param_t *param)`
  
  Callback to process incoming Active Message sent by `ucp_am_send_nb` routine.

- `typedef struct ucp_ep_params ucp_ep_params_t`
  
  Tuning parameters for the UCP endpoint.

Enumerations

- `enum ucp_ep_params_field {
    UCP_EP_PARAM_FIELD_REMOTE_ADDRESS = UCS_BIT(0),
    UCP_EP_PARAM_FIELD_ERR_HANDLING_MODE = UCS_BIT(1),
    UCP_EP_PARAM_FIELD_ERR_HANDLER = UCS_BIT(2),
    UCP_EP_PARAM_FIELD_USER_DATA = UCS_BIT(3),
    UCP_EP_PARAM_FIELD_SOCK_ADDR = UCS_BIT(4),
    UCP_EP_PARAM_FIELD_FLAGS = UCS_BIT(5),
    UCP_EP_PARAM_FIELD_CONN_REQUEST = UCS_BIT(6)
} `
  
  UCP endpoint parameters field mask.

- `enum ucp_ep_params_flags_field {
    UCP_EP_PARAMS_FLAGS_CLIENT_SERVER = UCS_BIT(0),
    UCP_EP_PARAMS_FLAGS_NO_LOOPBACK = UCS_BIT(1)
} `
  
  UCP endpoint parameters flags.

- `enum ucp_close_flags_t {
    UCP_EP_CLOSE_FLAG_FORCE = UCS_BIT(0)
} `
  
  Close UCP endpoint modes.

- `enum ucp_close_mode_t {
    UCP_EP_CLOSE_MODE_FORCE = 0,
    UCP_EP_CLOSE_MODE_FLUSH = 1
} `
  
  Close UCP endpoint modes.

- `enum ucp_cb_param_flags { UCP_CB_PARAM_FLAG_DATA = UCS_BIT(0) }
  
  Descriptor flags for Active Message callback.

- `enum ucp_err_handling_mode_t { UCP_ERR_HANDLING_MODE_NONE, UCP_ERR_HANDLING_MODE_PEER }
  
  Error handling mode for the UCP endpoint.
Functions

- `ucs_status_t ucp_ep_create (ucp_worker_h worker, const ucp_ep_params_t *params, ucp_ep_h *ep_p)`
  
  Create and connect an endpoint.

- `ucs_status_ptr_t ucp_ep_close_nb (ucp_ep_h ep, unsigned mode)`
  
  Non-blocking endpoint closure.

- `ucs_status_ptr_t ucp_ep_close_nbx (ucp_ep_h ep, const ucp_request_param_t *param)`
  
  Non-blocking endpoint closure.

- `void ucp_ep_print_info (ucp_ep_h ep, FILE *stream)`
  
  Print endpoint information.

- `ucs_status_ptr_t ucp_ep_flush_nb (ucp_ep_h ep, unsigned flags, ucp_send_callback_t cb)`
  
  Non-blocking flush of outstanding AMO and RMA operations on the endpoint.

- `ucs_status_ptr_t ucp_ep_close_nbx (ucp_ep_h ep, const ucp_request_param_t *param)`
  
  Non-blocking flush of outstanding AMO and RMA operations on the endpoint.

- `void ucp_request_release (void *request)`

- `void ucp_ep_destroy (ucp_ep_h ep)`

- `ucs_status_ptr_t ucp_disconnect_nb (ucp_ep_h ep)`

- `ucs_status_t ucp_request_test (void *request, ucp_tag_recv_info_t *info)`

- `ucs_status_t ucp_ep_flush (ucp_ep_h ep)`

- `ucs_status_ptr_t ucp_ep_modify_nb (ucp_ep_h ep, const ucp_ep_params_t *params)`
  
  Modify endpoint parameters.

6.6.1 Detailed Description

UCP Endpoint routines

6.6.2 Data Structure Documentation

6.6.2.1 struct ucp_stream_poll_ep

The structure defines the endpoint and its user data.

Data Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ucp_ep_h</td>
<td>Endpoint handle.</td>
</tr>
<tr>
<td>void *ep</td>
<td>User data associated with an endpoint passed in ucp_ep_params_t::user_data.</td>
</tr>
<tr>
<td>unsigned</td>
<td>Reserved for future use.</td>
</tr>
<tr>
<td>flags</td>
<td>Reserved for future use.</td>
</tr>
<tr>
<td>uint8_t</td>
<td>Reserved for future use.</td>
</tr>
</tbody>
</table>

6.6.2.2 struct ucp_ep_params

The structure defines the parameters that are used for the UCP endpoint tuning during the UCP ep creation.

Examples

- `ucp_client_server.c`, and `ucp_hello_world.c`
Data Fields

<table>
<thead>
<tr>
<th>Field Type</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>field_mask</td>
<td>uint64_t</td>
<td>Mask of valid fields in this structure, using bits from <code>ucp_ep_params_field</code>. Fields not specified in this mask will be ignored. Provides ABI compatibility with respect to adding new fields.</td>
</tr>
<tr>
<td>address</td>
<td>const ucp_address_t*</td>
<td>Destination address; this field should be set along with its corresponding bit in the field_mask - <code>UCP_EP_PARAM_FIELD_REMOTE_ADDRESS</code> and must be obtained using <code>ucp_worker_get_address</code>.</td>
</tr>
<tr>
<td>err_mode</td>
<td>ucp_err_handling_mode_t</td>
<td>Desired error handling mode, optional parameter. Default value is <code>UCP_ERR_HANDLING_MODE_NONE</code>.</td>
</tr>
<tr>
<td>err_handler</td>
<td>ucp_err_handler_t</td>
<td>Handler to process transport level failure.</td>
</tr>
<tr>
<td>user_data</td>
<td>void*</td>
<td>User data associated with an endpoint. See <code>ucp_stream_poll_ep_t</code> and <code>ucp_err_handler_t</code>.</td>
</tr>
<tr>
<td>sockaddr</td>
<td>ucs_sock_addr_t</td>
<td>Destination address in the form of a sockaddr; this field should be set along with its corresponding bit in the field_mask - <code>UCP_EP_PARAM_FIELD_SOCK_ADDR</code> and must be obtained from the user, it means that this type of the endpoint creation is possible only on client side in client-server connection establishment flow.</td>
</tr>
<tr>
<td>conn_request</td>
<td>ucp_conn_request_h</td>
<td>Connection request from client; this field should be set along with its corresponding bit in the field_mask - <code>UCP_EP_PARAM_FIELD_CONN_REQUEST</code> and must be obtained from <code>ucp_listener_conn_callback_t</code>, it means that this type of the endpoint creation is possible only on server side in client-server connection establishment flow.</td>
</tr>
</tbody>
</table>

6.6.3 Typedef Documentation

6.6.3.1 ucp_stream_poll_ep_t

typedef struct ucp_stream_poll_ep_t ucp_stream_poll_ep_t

The structure defines the endpoint and its user data.

6.6.3.2 ucp_ep_h

typedef struct ucp_ep_h ucp_ep_h

The endpoint handle is an opaque object that is used to address a remote worker. It typically provides a description of source, destination, or both. All UCP communication routines address a destination with the endpoint handle. The endpoint handle is associated with only one UCP context. UCP provides the endpoint create routine to create the endpoint handle and the destroy routine to destroy the endpoint handle.

6.6.3.3 ucp_conn_request_h

typedef struct ucp_conn_request_h ucp_conn_request_h

© 2021 Unified Communication X (UCX). All rights reserved.
A server-side handle to incoming connection request. Can be used to create an endpoint which connects back to the client.

6.6.3.4 ucp_am_callback_t

typedef ucs_status_t(\* ucp_am_callback_t) (void \*arg, void \*data, size_t length, ucp_ep_h reply_ep, unsigned flags)

When the callback is called, flags indicates how data should be handled.

Parameters

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>arg</td>
<td>User-defined argument.</td>
</tr>
<tr>
<td>in</td>
<td>data</td>
<td>Points to the received data. This data may persist after the callback returns and needs to be freed with ucp_am_data_release.</td>
</tr>
<tr>
<td>in</td>
<td>length</td>
<td>Length of data.</td>
</tr>
<tr>
<td>in</td>
<td>reply_ep</td>
<td>If the Active Message is sent with the UCP_AM_SEND_REPLY flag, the sending ep will be passed in. If not, NULL will be passed.</td>
</tr>
<tr>
<td>in</td>
<td>flags</td>
<td>If this flag is set to UCP_CB_PARAM_FLAG_DATA, the callback can return UCS_INPROGRESS and data will persist after the callback returns.</td>
</tr>
</tbody>
</table>

Returns

UCS_OK data will not persist after the callback returns.
UCS_INPROGRESS Can only be returned if flags is set to UCP_CB_PARAM_FLAG_DATA. If UCP_INPROGRESS is returned, data will persist after the callback has returned. To free the memory, a pointer to the data must be passed into ucp_am_data_release.

Note

This callback should be set and released by ucp_worker_set_am_handler function.

6.6.3.5 ucp_am_recv_callback_t

typedef ucs_status_t(\* ucp_am_recv_callback_t) (void \*arg, const void \*header, size_t header_length, void \*data, size_t length, const ucp_am_recv_param_t \*param)

The callback is always called from the progress context, therefore calling ucp_worker_progress() is not allowed. It is recommended to define callbacks with relatively short execution time to avoid blocking of communication progress.

Parameters

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>arg</td>
<td>User-defined argument.</td>
</tr>
<tr>
<td>in</td>
<td>header</td>
<td>User defined active message header. Can be NULL.</td>
</tr>
<tr>
<td>in</td>
<td>header_length</td>
<td>Active message header length in bytes. If this value is 0, the header pointer is undefined and should not be accessed.</td>
</tr>
<tr>
<td>in</td>
<td>data</td>
<td>Points to the received data if UCP_AM_RECV_ATTR_FLAG_RNDV flag is not set in ucp_am_recv_param_t:recv_attr. Otherwise it points to the internal UCP descriptor which can further be used for initiating data receive by using ucp_am_recv_data_nbx routine.</td>
</tr>
<tr>
<td>in</td>
<td>length</td>
<td>Length of data. If UCP_AM_RECV_ATTR_FLAG_RNDV flag is set in ucp_am_recv_param_t:recv_attr, it indicates the required receive buffer size for initiating rendezvous protocol.</td>
</tr>
<tr>
<td>in</td>
<td>param</td>
<td>Data receive parameters.</td>
</tr>
</tbody>
</table>
Returns

UCS_OK data will not persist after the callback returns. If UCP_AM_RECV_ATTR_FLAG_RNDV flag is set in `param->recv_attr`, the data descriptor will be dropped and the corresponding `ucp_am_send_nbx` call should complete with UCS_OK status.

UCS_INPROGRESS Can only be returned if `param->recv_attr` flags contains UCP_AM_RECV_ATTR_FLAG_RNDV. The data descriptor will persist after the callback has returned. To free the memory, a pointer to the data must be passed into `ucp_am_data_release` or, in the case of rendezvous descriptor, data receive is initiated by `ucp_am_recv_data_nbx`.

otherwise Can only be returned if `param->recv_attr` contains UCP_AM_RECV_ATTR_FLAG_RNDV. In this case data descriptor data will be dropped and the corresponding `ucp_am_send_nbx` call should complete with the status returned from the callback.

Note

This callback should be set and released by `ucp_worker_set_am_recv_handler` function.

6.6.3.6 ucp_ep_params_t

typedef struct ucp_ep_params ucp_ep_params_t

The structure defines the parameters that are used for the UCP endpoint tuning during the UCP ep creation.

6.6.4 Enumeration Type Documentation

6.6.4.1 ucp_ep_params_field

enum ucp_ep_params_field

The enumeration allows specifying which fields in `ucp_ep_params_t` are present. It is used to enable backward compatibility support.

Enumerator

<table>
<thead>
<tr>
<th>UCP_EP_PARAM_FIELD_REMOTE_ADDRESS</th>
<th>Address of remote peer</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCP_EP_PARAM_FIELD_ERR_HANDLING_MODE</td>
<td>Error handling mode. <code>ucp_err_handling_mode_t</code></td>
</tr>
<tr>
<td>UCP_EP_PARAM_FIELD_ERR_HANDLER</td>
<td>Handler to process transport level errors</td>
</tr>
<tr>
<td>UCP_EP_PARAM_FIELD_USER_DATA</td>
<td>User data pointer</td>
</tr>
<tr>
<td>UCP_EP_PARAM_FIELD_SOCK_ADDR</td>
<td>Socket address field</td>
</tr>
<tr>
<td>UCP_EP_PARAM_FIELD_FLAGS</td>
<td>Endpoint flags</td>
</tr>
<tr>
<td>UCP_EP_PARAM_FIELD_CONN_REQUEST</td>
<td>Connection request field</td>
</tr>
</tbody>
</table>

6.6.4.2 ucp_ep_params_flags_field

enum ucp_ep_params_flags_field

The enumeration list describes the endpoint's parameters flags supported by `ucp_ep_create()` function.
### 6.6 UCP Endpoint

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>UCP_EP_PARAMS_FLAGS_CLIENT_SERVER</code></td>
<td>Using a client-server connection establishment mechanism. <code>ucs_sock_addr_t sockaddr</code> field must be provided and contain the address of the remote peer.</td>
</tr>
<tr>
<td><code>UCP_EP_PARAMS_FLAGS_NO_LOOPBACK</code></td>
<td>Avoid connecting the endpoint to itself when connecting the endpoint to the same worker it was created on. Affects protocols which send to a particular remote endpoint, for example stream.</td>
</tr>
</tbody>
</table>

### 6.6.4.3 `ucp_ep_close_flags_t`

`enum ucp_ep_close_flags_t`

The enumeration is used to specify the behavior of `ucp_ep_close_nb`.

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Description</th>
</tr>
</thead>
</table>
| `UCP_EP_CLOSE_FLAG_FORCE` | `ucp_ep_close_nb` releases the endpoint without any confirmation from the peer. All outstanding requests will be completed with `UCS_ERR_CANCELED` error.  
**Note**  
This mode may cause transport level errors on remote side, so it requires set `UCP_ERR_HANDLING_MODE_PEER` for all endpoints created on both (local and remote) sides to avoid undefined behavior. If this flag is not set then `ucp_ep_close_nb` schedules flushes on all outstanding operations. |

### 6.6.4.4 `ucp_ep_close_mode`

`enum ucp_ep_close_mode`

The enumeration is used to specify the behavior of `ucp_ep_close_nb`.

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Description</th>
</tr>
</thead>
</table>
| `UCP_EP_CLOSE_MODE_FORCE` | `ucp_ep_close_nb` releases the endpoint without any confirmation from the peer. All outstanding requests will be completed with `UCS_ERR_CANCELED` error.  
**Note**  
This mode may cause transport level errors on remote side, so it requires set `UCP_ERR_HANDLING_MODE_PEER` for all endpoints created on both (local and remote) sides to avoid undefined behavior. |
| `UCP_EP_CLOSE_MODE_FLUSH` | `ucp_ep_close_nb` schedules flushes on all outstanding operations. |
6.6.4.5 ucp_cb_param_flags

enum ucp_cb_param_flags

In a callback, if flags is set to UCP_CB_PARAM_FLAG_DATA in a callback then data was allocated, so if UCS_INPROGRESS is returned from the callback, the data parameter will persist and the user has to call ucp_am_data_release when data is no longer needed.

Enumerator

| UCP_CB_PARAM_FLAG_DATA |

6.6.4.6 ucp_err_handling_mode_t

enum ucp_err_handling_mode_t

Specifies error handling mode for the UCP endpoint.

Enumerator

| UCP_ERR_HANDLING_MODE_NONE | No guarantees about error reporting, imposes minimal overhead from a performance perspective.
|                           | Note
|                           |     In this mode, any error reporting will not generate calls to ucp_ep_params_t::err_handler.
| UCP_ERR_HANDLING_MODE_PEER | Guarantees that send requests are always completed (successfully or error) even in case of remote failure, disables protocols and APIs which may cause a hang or undefined behavior in case of peer failure, may affect performance and memory footprint |

6.6.5 Function Documentation

6.6.5.1 ucp_ep_create()

ucs_status_t ucp_ep_create ( 
    ucp_worker_h worker,
    const ucp_ep_params_t * params,
    ucp_ep_h * ep_p )

This routine creates and connects an endpoint on a local worker for a destination address that identifies the remote worker. This function is non-blocking, and communications may begin immediately after it returns. If the connection process is not completed, communications may be delayed. The created endpoint is associated with one and only one worker.

Parameters

| in  | worker | Handle to the worker; the endpoint is associated with the worker. |
| in  | params | User defined ucp_ep_params_t configurations for the UCP endpoint. |
| out | ep_p   | A handle to the created endpoint. |
Returns

Error code as defined by ucs_status_t

Note

One of the following fields has to be specified:

- ucp_ep_params_t::address
- ucp_ep_params_t::sockaddr
- ucp_ep_params_t::conn_request

By default, ucp_ep_create() will connect an endpoint to itself if the endpoint is destined to the same worker on which it was created, i.e. params.address belongs to worker. This behavior can be changed by passing the UCP_EP_PARAMS_FLAGS_NO_LOOPBACK flag in params.flags. In that case, the endpoint will be connected to the next endpoint created in the same way on the same worker.

Examples

ucp_client_server.c, and ucp_hello_world.c.

6.6.5.2 ucp_ep_close_nb()

ucs_status_ptr_t ucp_ep_close_nb ( 
   ucp_ep_h ep, 
   unsigned mode )

This routine releases the endpoint. The endpoint closure process depends on the selected mode.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>ep</th>
<th>Handle to the endpoint to close.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>mode</td>
<td>One from ucp_ep_close_mode value.</td>
</tr>
</tbody>
</table>

Returns

UCS_OK - The endpoint is closed successfully.
UCS_PTR_IS_ERR(_ptr) - The closure failed and an error code indicates the transport level status. However, resources are released and the endpoint can no longer be used.
otherwise - The closure process is started, and can be completed at any point in time. A request handle is returned to the application in order to track progress of the endpoint closure. The application is responsible for releasing the handle using the ucp_request_free routine.

Note

ucp_ep_close_nb replaces deprecated ucp_disconnect_nb and ucp_ep_destroy

6.6.5.3 ucp_ep_close_nbx()

ucs_status_ptr_t ucp_ep_close_nbx ( 
   ucp_ep_h ep, 
   const ucp_request_param_t * param )
Parameters

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>ep</td>
<td>Handle to the endpoint to close.</td>
</tr>
<tr>
<td>in</td>
<td>param</td>
<td>Operation parameters, see ucp_request_param_t. This operation supports specific flags, which can be passed in param by ucp_request_param_t::flags. The exact set of flags is defined by ucp_ep_close_flags_t.</td>
</tr>
</tbody>
</table>

Returns

NULL - The endpoint is closed successfully.
UCS_PTR_IS_ERR(_ptr) - The closure failed and an error code indicates the transport level status. However, resources are released and the endpoint can no longer be used.
otherwise - The closure process is started, and can be completed at any point in time. A request handle is returned to the application in order to track progress of the endpoint closure.

Examples

ucp_client_server.c.

6.6.5.4 ucp_ep_print_info()

void ucp_ep_print_info (  
    ucp_ep_h ep,  
    FILE * stream )

This routine prints information about the endpoint transport methods, their thresholds, and other useful information associated with the endpoint.

Parameters

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>ep</td>
<td>Endpoint object whose configuration to print.</td>
</tr>
<tr>
<td>in</td>
<td>stream</td>
<td>Output stream to print the information to.</td>
</tr>
</tbody>
</table>

6.6.5.5 ucp_ep_flush_nb()

ucs_status_ptr_t ucp_ep_flush_nb (  
    ucp_ep_h ep,  
    unsigned flags,  
    ucp_send_callback_t cb )

This routine flushes all outstanding AMO and RMA communications on the endpoint. All the AMO and RMA operations issued on the ep prior to this call are completed both at the origin and at the target endpoint when this call returns.

Parameters

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>ep</td>
<td>UCP endpoint.</td>
</tr>
<tr>
<td>in</td>
<td>flags</td>
<td>Flags for flush operation. Reserved for future use.</td>
</tr>
<tr>
<td>in</td>
<td>cb</td>
<td>Callback which will be called when the flush operation completes.</td>
</tr>
</tbody>
</table>
6.6 UCP Endpoint

Returns

NULL - The flush operation was completed immediately.
UCS_PTR_IS_ERR(_ptr) - The flush operation failed.
otherwise - Flush operation was scheduled and can be completed in any point in time. The request handle is returned to the application in order to track progress. The application is responsible for releasing the handle using `ucp_request_free()` routine.

The following example demonstrates how blocking flush can be implemented using non-blocking flush:
```c
void empty_function(void *request, ucs_status_t status) {
}

ucs_status_t blocking_ep_flush(ucp_ep_h ep, ucp_worker_h worker) {
    void *request;
    request = ucp_ep_flush_nb(ep, 0, empty_function);
    if (request == NULL) {
        return UCS_OK;
    } else if (UCS_PTR_IS_ERR(request)) {
        return UCS_PTR_STATUS(request);
    } else {
        ucs_status_t status;
        do {
            ucp_worker_progress(worker);
            status = ucp_request_check_status(request);
        } while (status == UCS_INPROGRESS);
        ucp_request_free(request);
        return status;
    }
}
```

6.6.5.6 ucp_ep_flush_nbx()

```c
ucs_status_ptr_t ucp_ep_flush_nbx (ucp_ep_h ep,
const ucp_request_param_t * param)
```

This routine flushes all outstanding AMO and RMA communications on the endpoint. All the AMO and RMA operations issued on the `ep` prior to this call are completed both at the origin and at the target endpoint when this call returns.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>ep</th>
<th>UCP endpoint.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>param</td>
<td>Operation parameters, see <code>ucp_request_param_t</code>.</td>
</tr>
</tbody>
</table>

Returns

NULL - The flush operation was completed immediately.
UCS_PTR_IS_ERR(_ptr) - The flush operation failed.
otherwise - Flush operation was scheduled and can be completed in any point in time. The request handle is returned to the application in order to track progress.

Examples

ucp_hello_world.c.

6.6.5.7 ucp_request_release()

```c
void ucp_request_release ( void * request )
```
**Deprecated** Replaced by `ucp_request_free`.

**Examples**
```
ucp_hello_world.c.
```

### 6.6.5.8 `ucp_ep_destroy()`

```c
void ucp_ep_destroy (ucp_ep_h ep)
```

**Deprecated** Replaced by `ucp_ep_close_nb`.

**Examples**
```
ucp_hello_world.c.
```

### 6.6.5.9 `ucp_disconnect_nb()`

```c
ucp_status_ptr_t ucp_disconnect_nb (ucp_ep_h ep)
```

**Deprecated** Replaced by `ucp_ep_close_nb`.

### 6.6.5.10 `ucp_request_test()`

```c
ucp_status_t ucp_request_test (void *request, ucp_tag_recv_info_t *info)
```

**Deprecated** Replaced by `ucp_tag_recv_request_test` and `ucp_request_check_status` depends on use case.

**Note**
Please use `ucp_request_check_status` for cases that only need to check the completion status of an outstanding request. `ucp_request_check_status` can be used for any type of request. `ucp_tag_recv_request_test` should only be used for requests returned by `ucp_tag_recv_nb` (or request allocated by user for `ucp_tag_recv_nbr`) for which additional information (returned via the `info` pointer) is needed.

### 6.6.5.11 `ucp_ep_flush()`

```c
ucp_status_t ucp_ep_flush (ucp_ep_h ep)
```

**Deprecated** Replaced by `ucp_ep_flush_nb`.

© 2021 Unified Communication X (UCX). All rights reserved.
6.6.5.12 ucp_ep_modify_nb()

```c
ucp_status_ptr_t ucp_ep_modify_nb(
    ucp_ep_h ep,
    const ucp_ep_params_t * params)
```

**Deprecated** Use `ucp_listener_conn_handler_t` instead of `ucp_listener_accept_handler_t`, if you have other use case please submit an issue on [https://github.com/openucx/ucx](https://github.com/openucx/ucx) or report to [ucx-group@elist.ornl.gov](mailto:ucx-group@elist.ornl.gov)

This routine modifies endpoint created by `ucp_ep_create` or `ucp_listener_accept_callback_t`. For example, this API can be used to setup custom parameters like `ucp_ep_params_t::user_data` or `ucp_ep_params_t::err_handler` to endpoint created by `ucp_listener_accept_callback_t`.

**Parameters**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ep</code></td>
<td>A handle to the endpoint.</td>
</tr>
<tr>
<td><code>params</code></td>
<td>User defined <code>ucp_ep_params_t</code> configurations for the UCP endpoint.</td>
</tr>
</tbody>
</table>

**Returns**

- `NULL` - The endpoint is modified successfully.
- `UCS_PTR_IS_ERR(ptr)` - The reconfiguration failed and an error code indicates the status. However, the endpoint is not modified and can be used further.
- otherwise - The reconfiguration process is started, and can be completed at any point in time. A request handle is returned to the application in order to track progress of the endpoint modification. The application is responsible for releasing the handle using the `ucp_request_free` routine.

**Note**

See the documentation of `ucp_ep_params_t` for details, only some of the parameters can be modified.
6.7 UCP Communication routines

Data Structures

- struct ucp_err_handler
  UCP endpoint error handling context. More...

Typedefs

- typedef uint64_t ucp_tag_t
  UCP Tag Identifier.
- typedef struct ucp_recv_desc * ucp_tag_message_h
  UCP Message descriptor.
- typedef uint64_t ucp_datatype_t
  UCP Datatype Identifier.
- typedef void(∗ucp_send_callback_t)(void ∗request, ucs_status_t status)
  Completion callback for non-blocking sends.
- typedef void(∗ucp_send_nbx_callback_t)(void ∗request, ucs_status_t status, void ∗user_data)
  Completion callback for non-blocking sends ucp_tag_send_nbx call.
- typedef void(∗ucp_err_handler_cb_t)(void ∗arg, ucp_ep_h ep, ucs_status_t status)
  Callback to process peer failure.
- typedef struct ucp_err_handler ucp_err_handler_t
  UCP endpoint error handling context.
- typedef void(∗ucp_stream_recv_callback_t)(void ∗request, ucs_status_t status, size_t length)
  Completion callback for non-blocking stream oriented receives.
- typedef void(∗ucp_stream_recv_nbx_callback_t)(void ∗request, ucs_status_t status, size_t length, void ∗user_data)
  Completion callback for non-blocking stream receives ucp_stream_recv_nbx call.
- typedef void(∗ucp_tag_recv_callback_t)(void ∗request, ucs_status_t status, ucp_tag_recv_info_t ∗info)
  Completion callback for non-blocking tag receives.
- typedef void(∗ucp_tag_recv_nbx_callback_t)(void ∗request, ucs_status_t status, const ucp_tag_recv_info_t ∗tag_info, void ∗user_data)
  Completion callback for non-blocking tag receives ucp_tag_recv_nbx call.
- typedef void(∗ucp_am_recv_data_nbx_callback_t)(void ∗request, ucs_status_t status, size_t length, void ∗user_data)
  Completion callback for non-blocking Active Message receives.

Enumerations

- enum ucp_atomic_post_op_t {
  UCP_ATOMIC_POST_OP_ADD, UCP_ATOMIC_POST_OP_AND, UCP_ATOMIC_POST_OP_OR, 
  UCP_ATOMIC_POST_OP_XOR, 
  UCP_ATOMIC_POST_OP_LAST }
  Atomic operation requested for ucp_atomic_post.
- enum ucp_atomic_fetch_op_t {
  UCP_ATOMIC_FETCH_OP_FADD, UCP_ATOMIC_FETCH_OP_SWAP, UCP_ATOMIC_FETCH_OP_CSWAP, 
  UCP_ATOMIC_FETCH_OP_FAND, 
  UCP_ATOMIC_FETCH_OP_FOR, UCP_ATOMIC_FETCH_OP_FXOR, UCP_ATOMIC_FETCH_OP_LAST }
  Atomic operation requested for ucp_atomic_fetch.
6.7 UCP Communication routines

- **enum ucp_atomic_op_t**

  UCP_ATOMIC_OP_ADD, UCP_ATOMIC_OP_SWAP, UCP_ATOMIC_OP_CSWAP, UCP_ATOMIC_OP_AND,
  UCP_ATOMIC_OP_OR, UCP_ATOMIC_OP_XOR, UCP_ATOMIC_OP_LAST

  Atomic operation requested for ucp_atomic_op_nb.

- **enum ucp_stream_recv_flags_t**

  UCP_STREAM_RECV_FLAG_WAITALL = UCS_BIT(0)

  Flags to define behavior of ucp_stream_recv_nb function.

- **enum ucp_op_attr_t**

  UCP_OP_ATTR_FIELD_REQUEST = UCS_BIT(0), UCP_OP_ATTR_FIELD_CALLBACK = UCS_BIT(1),
  UCP_OP_ATTR_FIELD_USER_DATA = UCS_BIT(2), UCP_OP_ATTR_FIELD_DATATYPE = UCS_BIT(3),
  UCP_OP_ATTR_FIELD_FLAGS = UCS_BIT(4), UCP_OP_ATTR_FIELD_REPLY_BUFFER = UCS_BIT(5),
  UCP_OP_ATTR_FIELD_MEMORY_TYPE = UCS_BIT(6), UCP_OP_ATTR_FIELD_RECV_INFO = UCS_BIT(7),
  UCP_OP_ATTR_FLAG_NO_IMM_CMPL = UCS_BIT(16), UCP_OP_ATTR_FLAG_FAST_CMPL = UCS_BIT(17),
  UCP_OP_ATTR_FLAG_FORCE_IMM_CMPL = UCS_BIT(18)

  UCP operation fields and flags.

- **enum ucp_am_recv_attr_t**

  UCP_AM_RECV_ATTR_FIELD_REPLY_EP = UCS_BIT(0), UCP_AM_RECV_ATTR_FLAG_DATA = UCS_BIT(16),
  UCP_AM_RECV_ATTR_FLAG_RNDV = UCS_BIT(17)

  UCP AM receive data parameter fields and flags.

- **enum ucp_am_handler_param_field**

  UCP_AM_HANDLER_PARAM_FIELD_ID = UCS_BIT(0), UCP_AM_HANDLER_PARAM_FIELD_FLAGS = UCS_BIT(1),
  UCP_AM_HANDLER_PARAM_FIELD_CB = UCS_BIT(2), UCP_AM_HANDLER_PARAM_FIELD_ARG = UCS_BIT(3)

  UCP AM receive data parameters fields and flags.

**Functions**

- **ucs_status_ptr_t ucp_am_send_nb**( ep: ucp_ep_h, id: uint16_t, buffer: const void *, count: size_t, datatype: ucp_datatype_t, cb: ucp_send_callback_t, flags: unsigned)

  Send Active Message.

- **ucs_status_ptr_t ucp_am_send_nbx**( ep: ucp_ep_h, id: unsigned, header: const void *, header_length: size_t, buffer: const void *, count: size_t, param: const ucp_request_param_t *)

  Send Active Message.

- **ucs_status_ptr_t ucp_am_recv_data_nbx**( worker: ucp_worker_h, data_desc: void *, buffer: const void *, count: size_t, param: const ucp_request_param_t *)

  Receive Active Message sent with rendezvous protocol.

- **void ucp_am_data_release**( worker: ucp_worker_h, data: void *)

  Releases Active Message data.

- **ucs_status_ptr_t ucp_stream_send_nb**( ep: ucp_ep_h, buffer: const void *, count: size_t, datatype: ucp_datatype_t, cb: ucp_send_callback_t, flags: unsigned)

  Non-blocking stream send operation.

- **ucs_status_ptr_t ucp_stream_send_nbx**( ep: ucp_ep_h, buffer: const void *, count: size_t, param: const ucp_request_param_t *)

  Non-blocking stream send operation.

- **ucs_status_ptr_t ucp_tag_send_nb**( ep: ucp_ep_h, buffer: const void *, count: size_t, datatype: ucp_datatype_t, tag: ucp_tag_t, cb: ucp_send_callback_t)

  Non-blocking tagged-send operations.

- **ucs_status_t ucp_tag_send_nbr**( ep: ucp_ep_h, buffer: const void *, count: size_t, datatype: ucp_datatype_t, tag: ucp_tag_t, req: void *)

  Non-blocking tagged-send operations with user provided request.

- **ucs_status_ptr_t ucp_tag_send_sync_nb**( ep: ucp_ep_h, buffer: const void *, count: size_t, datatype: ucp_datatype_t, tag: ucp_tag_t, cb: ucp_send_callback_t)

  Non-blocking synchronous tagged-send operation.

- **ucs_status_ptr_t ucp_tag_send_nbx**( ep: ucp_ep_h, buffer: const void *, count: size_t, tag: ucp_tag_t, param: const ucp_request_param_t *)

  Non-blocking synchronous tagged-send operation.

© 2021 Unified Communication X (UCX). All rights reserved.
Non-blocking tagged-send operation.

- `ucs_status_ptr_t ucp_tag_send_sync_nbx (ucp_ep_h ep, const void *buffer, size_t count, ucp_tag_t tag, const ucp_request_param_t *param)`

Non-blocking synchronous tagged-send operation.

- `ucs_status_ptr_t ucp_stream_recv_nb (ucp_ep_h ep, void *buffer, size_t count, ucp_datatype_t datatype, ucp_stream_recv_callback_t cb, size_t *length, unsigned flags)`

Non-blocking stream receive operation of structured data into a user-supplied buffer.

- `ucs_status_ptr_t ucp_stream_recv_nbx (ucp_ep_h ep, void *buffer, size_t count, size_t *length, const ucp_request_param_t *param)`

Non-blocking stream receive operation of structured data into a user-supplied buffer.

- `ucs_status_ptr_t ucp_stream_recv_data_nb (ucp_ep_h ep, size_t *length)`

Non-blocking stream receive operation of unstructured data into a UCP-supplied buffer.

- `ucs_status_ptr_t ucp_tag_recv_nb (ucp_worker_h worker, void *buffer, size_t count, ucp_datatype_t datatype, ucp_tag_t tag, ucp_tag_t tag_mask, ucp_tag_recv_callback_t cb)`

Non-blocking tagged-receive operation.

- `ucs_status_t ucp_tag_recv_nbr (ucp_worker_h worker, void *buffer, size_t count, ucp_datatype_t datatype, ucp_tag_t tag, ucp_tag_t tag_mask, void *req)`

Non-blocking tagged-receive operation.

- `ucs_status_ptr_t ucp_tag_recv_nbx (ucp_worker_h worker, void *buffer, size_t count, ucp_tag_t tag, ucp_tag_t tag_mask, const ucp_request_param_t *param)`

Non-blocking tagged-receive operation.

- `ucp_tag_message_h ucp_tag_probe_nb (ucp_worker_h worker, ucp_tag_t tag, ucp_tag_t tag_mask, int remove, ucp_tag_recv_info_t *info)`

Non-blocking probe and return a message.

- `ucs_status_ptr_t ucp_tag_msg_recv_nb (ucp_worker_h worker, void *buffer, size_t count, ucp_datatype_t datatype, ucp_tag_message_h message, ucp_tag_recv_callback_t cb)`

Non-blocking receive operation for a probed message.

- `ucs_status_ptr_t ucp_tag_msg_recv_nbx (ucp_worker_h worker, void *buffer, size_t count, ucp_tag_message_h message, const ucp_request_param_t *param)`

Non-blocking receive operation for a probed message.

- `ucs_status_t ucp_put_nbi (ucp_ep_h ep, const void *buffer, size_t length, uint64_t remote_addr, ucp_rkey_h rkey)`

Non-blocking implicit remote memory put operation.

- `ucs_status_ptr_t ucp_put_nb (ucp_ep_h ep, const void *buffer, size_t length, uint64_t remote_addr, ucp_rkey_h rkey, ucp_send_callback_t cb)`

Non-blocking remote memory put operation.

- `ucs_status_ptr_t ucp_put_nbx (ucp_ep_h ep, const void *buffer, size_t count, uint64_t remote_addr, ucp_rkey_h rkey, const ucp_request_param_t *param)`

Non-blocking remote memory put operation.

- `ucs_status_t ucp_get_nbi (ucp_ep_h ep, void *buffer, size_t length, uint64_t remote_addr, ucp_rkey_h rkey)`

Non-blocking implicit remote memory get operation.

- `ucs_status_ptr_t ucp_get_nb (ucp_ep_h ep, void *buffer, size_t length, uint64_t remote_addr, ucp_rkey_h rkey, ucp_send_callback_t cb)`

Non-blocking remote memory get operation.

- `ucs_status_ptr_t ucp_get_nbx (ucp_ep_h ep, void *buffer, size_t count, uint64_t remote_addr, ucp_rkey_h rkey, const ucp_request_param_t *param)`

Non-blocking remote memory get operation.

- `ucs_status_t ucp_atomic_post (ucp_ep_h ep, ucp_atomic_post_op_t opcode, uint64_t value, size_t op_size, uint64_t remote_addr, ucp_rkey_h rkey)`

Post an atomic memory operation.

- `ucs_status_ptr_t ucp_atomic_fetch_nb (ucp_ep_h ep, ucp_atomic_fetch_op_t opcode, uint64_t_t value, void *result, size_t op_size, uint64_t remote_addr, ucp_rkey_h rkey, ucp_send_callback_t cb)`

Post an atomic fetch operation.
6.7 UCP Communication routines

- `ucs_status_ptr_t ucp_atomic_op_nbx (ucp_ep_h ep, ucp_atomic_op_t opcode, const void *buffer, size_t count, uint64_t remote_addr, ucp_rkey_h rkey, const ucp_request_param_t *param)`
  Post an atomic memory operation.

- `ucs_status_t ucp_request_check_status (void *request)`
  Check the status of non-blocking request.

- `ucs_status_t ucp_tag_recv_request_test (void *request, ucp_tag_recv_info_t *info)`
  Check the status and currently available state of non-blocking request returned from ucp_tag_recv_nb routine.

- `ucs_status_t ucp_stream_recv_request_test (void *request, size_t *length_p)`
  Check the status and currently available state of non-blocking request returned from ucp_stream_recv_nb routine.

- `void ucp_request_cancel (ucp_worker_h worker, void *request)`
  Cancel an outstanding communications request.

- `void ucp_stream_data_release (ucp_ep_h ep, void *data)`
  Release UCP data buffer returned by ucp_stream_recv_data_nb.

- `void *ucp_request_free (void *request)`
  Release a communications request.

- `void *ucp_request_alloc (ucp_worker_h worker)`
  Create an empty communications request.

- `int ucp_request_is_completed (void *request)`

- `ucp_status_t ucp_put (ucp_ep_h ep, const void *buffer, size_t length, uint64_t remote_addr, ucp_rkey_h rkey)`
  Blocking remote memory put operation.

- `ucp_status_t ucp_get (ucp_ep_h ep, void *buffer, size_t length, uint64_t remote_addr, ucp_rkey_h rkey)`
  Blocking remote memory get operation.

- `ucp_status_t ucp_atomic_add32 (ucp_ep_h ep, uint32_t add, uint64_t remote_addr, ucp_rkey_h rkey)`
  Blocking atomic add operation for 32 bit integers.

- `ucp_status_t ucp_atomic_add64 (ucp_ep_h ep, uint64_t add, uint64_t remote_addr, ucp_rkey_h rkey)`
  Blocking atomic add operation for 64 bit integers.

- `ucp_status_t ucp_atomic_fadd32 (ucp_ep_h ep, uint32_t add, uint64_t remote_addr, ucp_rkey_h rkey, uint32_t *result)`
  Blocking atomic fetch and add operation for 32 bit integers.

- `ucp_status_t ucp_atomic_fadd64 (ucp_ep_h ep, uint64_t add, uint64_t remote_addr, ucp_rkey_h rkey, uint64_t *result)`
  Blocking atomic fetch and add operation for 64 bit integers.

- `ucp_status_t ucp_atomic_swap32 (ucp_ep_h ep, uint32_t swap, uint64_t remote_addr, ucp_rkey_h rkey, uint32_t *result)`
  Blocking atomic swap operation for 32 bit values.

- `ucp_status_t ucp_atomic_swap64 (ucp_ep_h ep, uint64_t swap, uint64_t remote_addr, ucp_rkey_h rkey, uint64_t *result)`
  Blocking atomic swap operation for 64 bit values.

- `ucp_status_t ucp_atomic_cswap32 (ucp_ep_h ep, uint32_t compare, uint32_t swap, uint64_t remote_addr, ucp_rkey_h rkey, uint32_t *result)`
  Blocking atomic conditional swap (cswap) operation for 32 bit values.

- `ucp_status_t ucp_atomic_cswap64 (ucp_ep_h ep, uint64_t compare, uint64_t swap, uint64_t remote_addr, ucp_rkey_h rkey, uint64_t *result)`
  Blocking atomic conditional swap (cswap) operation for 64 bit values.

6.7.1 Detailed Description

UCP Communication routines
6.7.2 Data Structure Documentation

6.7.2.1 struct ucp_err_handler

This structure should be initialized in ucp_ep_params_t to handle peer failure

Data Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ucp_err_handler_cb_t</td>
<td>cb</td>
<td>Error handler callback, if NULL, will not be called.</td>
</tr>
<tr>
<td>void * arg</td>
<td>arg</td>
<td>User defined argument associated with an endpoint, it will be overridden by</td>
</tr>
<tr>
<td></td>
<td>ucp_ep_params_t::user_data</td>
<td>if both are set.</td>
</tr>
</tbody>
</table>

6.7.3 Typedef Documentation

6.7.3.1 ucp_tag_t

typedef uint64_t ucp_tag_t

UCP tag identifier is a 64bit object used for message identification. UCP tag send and receive operations use the object for an implementation tag matching semantics (derivative of MPI tag matching semantics).

6.7.3.2 ucp_tag_message_h

typedef struct ucp_recv_desc * ucp_tag_message_h

UCP Message descriptor is an opaque handle for a message returned by ucp_tag Probe_nb. This handle can be passed to ucp_tag_msg_recv_nb in order to receive the message data to a specific buffer.

6.7.3.3 ucp_datatype_t

typedef uint64_t ucp_datatype_t

UCP datatype identifier is a 64bit object used for datatype identification. Predefined UCP identifiers are defined by ucp_dt_type.

6.7.3.4 ucp_send_callback_t

typedef void (* ucp_send_callback_t) (void *request, ucs_status_t status)

This callback routine is invoked whenever the send operation is completed. It is important to note that the call-back is only invoked in a case when the operation cannot be completed in place.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>request</td>
<td>The completed send request.</td>
</tr>
<tr>
<td>in status</td>
<td>completion status. If the send operation was completed successfully UCS_OK is returned. If send operation was canceled UCS_ERR_CANCELED is returned. Otherwise, an error status is returned.</td>
<td></td>
</tr>
</tbody>
</table>
6.7.3.5 ucp_send_nbx_callback_t

typedef void(* ucp_send_nbx_callback_t) (void *request, ucs_status_t status, void *user_data)

This callback routine is invoked whenever the send operation is completed. It is important to note that the call-back is only invoked in a case when the operation cannot be completed in place.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>request</th>
<th>The completed send request.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>status</td>
<td>Completion status. If the send operation was completed successfully UCS_OK is returned. If send operation was canceled UCS_ERR_CANCELED is returned. Otherwise, an error status is returned.</td>
</tr>
<tr>
<td>in</td>
<td>user_data</td>
<td>User data passed to &quot;user_data&quot; value, see ucp_request_param_t</td>
</tr>
</tbody>
</table>

Examples

ucp_client_server.c.

6.7.3.6 ucp_err_handler_cb_t

typedef void(* ucp_err_handler_cb_t) (void *arg, ucp_ep_h ep, ucs_status_t status)

This callback routine is invoked when transport level error detected.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>arg</th>
<th>User argument to be passed to the callback.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>ep</td>
<td>Endpoint to handle transport level error. Upon return from the callback, this ep is no longer usable and all subsequent operations on this ep will fail with the error code passed in status.</td>
</tr>
<tr>
<td>in</td>
<td>status</td>
<td>error status.</td>
</tr>
</tbody>
</table>

6.7.3.7 ucp_err_handler_t

typedef struct ucp_err_handler ucp_err_handler_t

This structure should be initialized in ucp_ep_params_t to handle peer failure

6.7.3.8 ucp_stream_recv_callback_t

typedef void(* ucp_stream_recv_callback_t) (void *request, ucs_status_t status, size_t length)

This callback routine is invoked whenever the receive operation is completed and the data is ready in the receive buffer.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>request</th>
<th>The completed receive request.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>status</td>
<td>Completion status. If the send operation was completed successfully UCS_OK is returned. Otherwise, an error status is returned.</td>
</tr>
<tr>
<td>in</td>
<td>length</td>
<td>The size of the received data in bytes, always boundary of base datatype size. The value is valid only if the status is UCS_OK.</td>
</tr>
</tbody>
</table>
6.7.3.9 ucp_stream_recv_nbx_callback_t

typedef void(* ucp_stream_recv_nbx_callback_t) (void *request, ucs_status_t status, size_t length, void *user_data)

This callback routine is invoked whenever the receive operation is completed and the data is ready in the receive buffer.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>request</td>
<td>The completed receive request.</td>
</tr>
<tr>
<td>status</td>
<td>Completion status. If the send operation was completed successfully UCS_OK is returned. Otherwise, an error status is returned.</td>
</tr>
<tr>
<td>length</td>
<td>The size of the received data in bytes, always on the boundary of base datatype size. The value is valid only if the status is UCS_OK.</td>
</tr>
<tr>
<td>user_data</td>
<td>User data passed to &quot;user_data&quot; value, see ucp_request_param_t.</td>
</tr>
</tbody>
</table>

6.7.3.10 ucp_tag_recv_callback_t

typedef void(* ucp_tag_recv_callback_t) (void *request, ucs_status_t status, ucp_tag_recv_info_t *info)

This callback routine is invoked whenever the receive operation is completed and the data is ready in the receive buffer.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>request</td>
<td>The completed receive request.</td>
</tr>
<tr>
<td>status</td>
<td>Completion status. If the send operation was completed successfully UCS_OK is returned. If send operation was canceled UCS_ERR_CANCELED is returned. If the data can not fit into the receive buffer the UCS_ERR_MESSAGE_TRUNCATED error code is returned. Otherwise, an error status is returned.</td>
</tr>
<tr>
<td>info</td>
<td>Completion information. The info descriptor is valid only if the status is UCS_OK.</td>
</tr>
</tbody>
</table>

6.7.3.11 ucp_tag_recv_nbx_callback_t

typedef void(* ucp_tag_recv_nbx_callback_t) (void *request, ucs_status_t status, const ucp_tag_recv_info_t *tag_info, void *user_data)

This callback routine is invoked whenever the receive operation is completed and the data is ready in the receive buffer.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>request</td>
<td>The completed receive request.</td>
</tr>
<tr>
<td>status</td>
<td>Completion status. If the receive operation was completed successfully UCS_OK is returned. If send operation was canceled, UCS_ERR_CANCELED is returned. If the data can not fit into the receive buffer the UCS_ERR_MESSAGE_TRUNCATED error code is returned. Otherwise, an error status is returned.</td>
</tr>
</tbody>
</table>
Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>info</th>
<th>Completion information The info descriptor is valid only if the status is UCS_OK.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>user_data</td>
<td>User data passed to &quot;user_data&quot; value, see ucp_request_param_t</td>
</tr>
</tbody>
</table>

6.7.3.12 ucp_am_recv_data_nbx_callback_t

typedef void(*ucp_am_recv_data_nbx_callback_t)(void *request, ucs_status_t status, size_t length, void *user_data)

This callback routine is invoked whenever the receive operation is completed and the data is ready in the receive buffer.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>request</th>
<th>The completed receive request.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>status</td>
<td>Completion status. If the receive operation was completed successfully UCS_OK is returned. Otherwise, an error status is returned.</td>
</tr>
<tr>
<td>in</td>
<td>length</td>
<td>The size of the received data in bytes, always boundary of base datatype size. The value is valid only if the status is UCS_OK.</td>
</tr>
<tr>
<td>in</td>
<td>user_data</td>
<td>User data passed to &quot;user_data&quot; value, see ucp_request_param_t</td>
</tr>
</tbody>
</table>

6.7.4 Enumeration Type Documentation

6.7.4.1 ucp_atomic_post_op_t

enum ucp_atomic_post_op_t

This enumeration defines which atomic memory operation should be performed by the ucp_atomic_post family of functions. All of these are non-fetching atomics and will not result in a request handle.

Enumerator

<table>
<thead>
<tr>
<th>UCP_ATOMIC_POST_OP_ADD</th>
<th>Atomic add</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCP_ATOMIC_POST_OP_AND</td>
<td>Atomic and</td>
</tr>
<tr>
<td>UCP_ATOMIC_POST_OP_OR</td>
<td>Atomic or</td>
</tr>
<tr>
<td>UCP_ATOMIC_POST_OP_XOR</td>
<td>Atomic xor</td>
</tr>
<tr>
<td>UCP_ATOMIC_POST_OP_LAST</td>
<td></td>
</tr>
</tbody>
</table>

6.7.4.2 ucp_atomic_fetch_op_t

enum ucp_atomic_fetch_op_t

This enumeration defines which atomic memory operation should be performed by the ucp_atomic_fetch family of functions. All of these functions will fetch data from the remote node.
Enumerations

<table>
<thead>
<tr>
<th>Enum Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCP_ATOMIC_FETCH_OP_FADD</td>
<td>Atomic Fetch and add</td>
</tr>
<tr>
<td>UCP_ATOMIC_FETCH_OP_SWAP</td>
<td>Atomic swap</td>
</tr>
<tr>
<td>UCP_ATOMIC_FETCH_OP_CSWAP</td>
<td>Atomic conditional swap</td>
</tr>
<tr>
<td>UCP_ATOMIC_FETCH_OP_FAND</td>
<td>Atomic Fetch and and</td>
</tr>
<tr>
<td>UCP_ATOMIC_FETCH_OP_FOR</td>
<td>Atomic Fetch and or</td>
</tr>
<tr>
<td>UCP_ATOMIC_FETCH_OP_FXOR</td>
<td>Atomic Fetch and xor</td>
</tr>
<tr>
<td>UCP_ATOMIC_FETCH_OP_LAST</td>
<td></td>
</tr>
</tbody>
</table>

6.7.4.3 ucp_atomic_op_t

enum ucp_atomic_op_t

This enumeration defines which atomic memory operation should be performed by the ucp_atomic_op_nbx routine.

Enumerations

<table>
<thead>
<tr>
<th>Enum Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCP_ATOMIC_OP_ADD</td>
<td>Atomic add</td>
</tr>
<tr>
<td>UCP_ATOMIC_OP_SWAP</td>
<td>Atomic swap</td>
</tr>
<tr>
<td>UCP_ATOMIC_OP_CSWAP</td>
<td>Atomic conditional swap</td>
</tr>
<tr>
<td>UCP_ATOMIC_OP_AND</td>
<td>Atomic and</td>
</tr>
<tr>
<td>UCP_ATOMIC_OP_OR</td>
<td>Atomic or</td>
</tr>
<tr>
<td>UCP_ATOMIC_OP_XOR</td>
<td>Atomic xor</td>
</tr>
<tr>
<td>UCP_ATOMIC_OP_LAST</td>
<td></td>
</tr>
</tbody>
</table>

6.7.4.4 ucp_stream_recv_flags_t

enum ucp_stream_recv_flags_t

This enumeration defines behavior of ucp_stream_recv_nb function.

Enumerations

<table>
<thead>
<tr>
<th>Enum Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCP_STREAM_RECV_FLAG_WAITALL</td>
<td>This flag requests that the operation will not be completed until all requested data is received and placed in the user buffer.</td>
<td></td>
</tr>
</tbody>
</table>

6.7.4.5 ucp_op_attr_t

enum ucp_op_attr_t

The enumeration allows specifying which fields in ucp_request_param_t are present and operation flags are used. It is used to enable backward compatibility support.

Enumerations

<table>
<thead>
<tr>
<th>Enum Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCP_OP_ATTR_FIELD_REQUEST</td>
<td>request field</td>
</tr>
</tbody>
</table>

© 2021 Unified Communication X (UCX). All rights reserved.
6.7 UCP Communication routines

### 6.7.4.6 ucp_am_recv_attr_t

**enum ucp_am_recv_attr_t**

The enumeration allows specifying which fields in `ucp_am_recv_param_t` are present and receive operation flags are used. It is used to enable backward compatibility support.

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCP_AM_RECV_ATTR_FIELD_REPLY_EP</td>
<td>reply_ep field</td>
</tr>
<tr>
<td>UCP_AM_RECV_ATTR_FLAG_DATA</td>
<td>Indicates that the data provided in <code>ucp_am_recv_callback_t</code> callback can be held by the user. If UCS_INPROGRESS is returned from the callback, the data parameter will persist and the user has to call <code>ucp_am_data_release</code> when data is no longer needed. This flag is mutually exclusive with <code>UCP_AM_RECV_ATTR_FLAG_RNDV</code>.</td>
</tr>
<tr>
<td>UCP_AM_RECV_ATTR_FLAG_RNDV</td>
<td>Indicates that the arriving data was sent using rendezvous protocol. In this case <code>data</code> parameter of the <code>ucp_am_recv_callback_t</code> points to the internal UCP descriptor, which can be used for obtaining the actual data by calling <code>ucp_am_recv_data_nbx</code> routine. This flag is mutually exclusive with <code>UCP_AM_RECV_ATTR_FLAG_DATA</code>.</td>
</tr>
</tbody>
</table>

---

### 6.7.4.7 ucp_am_handler_param_field

**enum ucp_am_handler_param_field**

The enumeration allows specifying which fields in `ucp_am_handler_param_t` are present. It is used to enable backward compatibility support.

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCP_AM_HANDLER_PARAM_FIELD_ID</td>
<td>Indicates that <code>ucp_am_handler_param_t::id</code> field is valid.</td>
</tr>
</tbody>
</table>
Enumerator

<table>
<thead>
<tr>
<th>Enumerator</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCP_AM_HANDLER_PARAM_FIELD_FLAGS</td>
</tr>
<tr>
<td>UCP_AM_HANDLER_PARAM_FIELD_CB</td>
</tr>
<tr>
<td>UCP_AM_HANDLER_PARAM_FIELD_ARG</td>
</tr>
</tbody>
</table>

6.7.5 Function Documentation

6.7.5.1 ucp_am_send_nb()

```c
ucs_status_ptr_t ucp_am_send_nb (
    ucp_ep_h ep,
    uint16_t id,
    const void * buffer,
    size_t count,
    ucp_datatype_t datatype,
    ucp_send_callback_t cb,
    unsigned flags
)
```

This routine sends an Active Message to an ep. It does not support CUDA memory.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>ep</th>
<th>UCP endpoint where the Active Message will be run.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>id</td>
<td>Active Message id. Specifies which registered callback to run.</td>
</tr>
<tr>
<td>in</td>
<td>buffer</td>
<td>Pointer to the data to be sent to the target node of the Active Message.</td>
</tr>
<tr>
<td>in</td>
<td>count</td>
<td>Number of elements to send.</td>
</tr>
<tr>
<td>in</td>
<td>datatype</td>
<td>Datatype descriptor for the elements in the buffer.</td>
</tr>
<tr>
<td>in</td>
<td>cb</td>
<td>Callback that is invoked upon completion of the data transfer if it is not completed immediately.</td>
</tr>
<tr>
<td>in</td>
<td>flags</td>
<td>Operation flags as defined by ucp_send_am_flags.</td>
</tr>
</tbody>
</table>

Returns

NULL Active Message was sent immediately.

UCS_PTR_IS_ERR(_ptr) Error sending Active Message.

otherwise Pointer to request, and Active Message is known to be completed after cb is run.

6.7.5.2 ucp_am_send_nbx()

```c
ucs_status_ptr_t ucp_am_send_nbx (
    ucp_ep_h ep,
    unsigned id,
    const void * header,
    size_t header_length,
    const void * buffer,
    size_t count,
    const ucp_request_param_t * param
)
```

This routine sends an Active Message to an ep. If the operation completes immediately, then the routine returns NULL and the callback function is ignored, even if specified. Otherwise, if no error is reported and a callback is
6.7 UCP Communication routines

requested (i.e. the UCP_OP_ATTR_FIELD_CALLBACK flag is set in the op_attr_mask field of param), then the UCP library will schedule invocation of the callback routine param->cb.send upon completion of the operation.

Note

If UCP_OP_ATTR_FLAG_NO_IMM_CMPL flag is set in the op_attr_mask field of param, then the operation will return a request handle, even if it completes immediately.

Currently Active Message API supports communication operations with host memory only.

This operation supports specific flags, which can be passed in param by ucp_request_param_t::flags. The exact set of flags is defined by ucp_send_am_flags.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>ep</th>
<th>UCP endpoint where the Active Message will be run.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>id</td>
<td>Active Message id. Specifies which registered callback to run.</td>
</tr>
<tr>
<td>in</td>
<td>header</td>
<td>User defined Active Message header. NULL value is allowed if no header needed. In this case header_length should be set to 0.</td>
</tr>
<tr>
<td>in</td>
<td>header_length</td>
<td>Active message header length in bytes.</td>
</tr>
<tr>
<td>in</td>
<td>buffer</td>
<td>Pointer to the data to be sent to the target node of the Active Message.</td>
</tr>
<tr>
<td>in</td>
<td>count</td>
<td>Number of elements to send.</td>
</tr>
<tr>
<td>in</td>
<td>param</td>
<td>Operation parameters, see ucp_request_param_t.</td>
</tr>
</tbody>
</table>

Note

Sending only header without actual data is allowed and is recommended for transferring latency-critical amount of data.

The maximum allowed header size can be obtained by querying worker attributes by ucp_worker_query routine.

Returns

NULL - Active Message was sent immediately.
UCS_PTR_IS_ERR(_ptr) - Error sending Active Message.
otherwise - Operation was scheduled for send and can be completed at any point in time. The request handle is returned to the application in order to track progress of the message. If user request was not provided in param->request, the application is responsible for releasing the handle using ucp_request_free routine.

Examples

ucp_client_server.c.

6.7.5.3 ucp_am_recv_data_nbx()

ucp_status_ptr_t ucp_am_recv_data_nbx ( 
    ucp_worker_h worker, 
    void * data_desc, 
    void * buffer, 
    size_t count, 
    const ucp_request_param_t * param )

This routine receives a message that is described by the data descriptor data_desc, local address buffer, size count and param parameters on the worker. The routine is non-blocking and therefore returns immediately. The receive operation is considered completed when the message is delivered to the buffer. If the receive operation cannot be started the routine returns an error.
Note

After this call UCP takes ownership of data_desc descriptor, so there is no need to release it even if the operation fails. The routine returns a request handle instead, which can further be used for tracking operation progress. Currently Active Message API supports communication operations with host memory only.

Parameters

| in  | worker       | Worker that is used for the receive operation. |
| in  | data_desc    | Data descriptor, provided in ucp_am_recv_callback_t routine. |
| in  | buffer       | Pointer to the buffer to receive the data. |
| in  | count        | Number of elements to receive into buffer. |
| in  | param        | Operation parameters, see ucp_request_param_t. |

Returns

NULL - The receive operation was completed immediately. In this case, if param->recv_info.length is specified in the param, the value to which it points is updated with the size of the received message.
UCS_PTR_IS_ERR(ptr) - The receive operation failed.
otherwise - Receive operation was scheduled and can be completed at any point in time. The request handle is returned to the application in order to track operation progress. If user request was not provided in param->request, the application is responsible for releasing the handle using ucp_request_free routine.

Examples

ucp_client_server.c.

6.7.5.4 ucp_am_data_release()

void ucp_am_data_release ( 
    ucp_worker_h worker, 
    void * data )

This routine releases data that persisted through an Active Message callback because that callback returned UCS_INPROGRESS.

Parameters

| in  | worker       | Worker which received the Active Message. |
| in  | data         | Pointer to data that was passed into the Active Message callback as the data parameter. |

6.7.5.5 ucp_stream_send_nb()

ucs_status_ptr_t ucp_stream_send_nb ( 
    ucp_ep_h ep, 
    const void * buffer, 
    size_t count, 
    ucp_datatype_t datatype, 
    ucp_send_callback_t cb, 
    unsigned flags )
This routine sends data that is described by the local address buffer, size count, and datatype object to the destination endpoint ep. The routine is non-blocking and therefore returns immediately, however the actual send operation may be delayed. The send operation is considered completed when it is safe to reuse the source buffer. If the send operation is completed immediately the routine returns UCS_OK and the callback function cb is not invoked. If the operation is not completed immediately and no error reported, then the UCP library will schedule invocation of the callback cb upon completion of the send operation. In other words, the completion of the operation will be signaled either by the return code or by the callback.

Note

The user should not modify any part of the buffer after this operation is called, until the operation completes.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>ep</th>
<th>Destination endpoint handle.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>buffer</td>
<td>Pointer to the message buffer (payload).</td>
</tr>
<tr>
<td>in</td>
<td>count</td>
<td>Number of elements to send.</td>
</tr>
<tr>
<td>in</td>
<td>datatype</td>
<td>Datatype descriptor for the elements in the buffer.</td>
</tr>
<tr>
<td>in</td>
<td>cb</td>
<td>Callback function that is invoked whenever the send operation is completed. It is important to note that the callback is only invoked in the event that the operation cannot be completed in place.</td>
</tr>
<tr>
<td>in</td>
<td>flags</td>
<td>Reserved for future use.</td>
</tr>
</tbody>
</table>

Returns

NULL - The send operation was completed immediately.
UCS_PTR_IS_ERR(_ptr) - The send operation failed.
otherwise - Operation was scheduled for send and can be completed in any point in time. The request handle is returned to the application in order to track progress of the message. The application is responsible for releasing the handle using ucp_request_free routine.

6.7.5.6 ucp_stream_send_nbx()

ucb_status_ptr_t ucp_stream_send_nbx (  
    ucp_ep_h ep,  
    const void * buffer,  
    size_t count,  
    const ucp_request_param_t * param )

This routine sends data that is described by the local address buffer, size count object to the destination endpoint ep. The routine is non-blocking and therefore returns immediately, however the actual send operation may be delayed. The send operation is considered completed when it is safe to reuse the source buffer. If the send operation is completed immediately the routine returns UCS_OK.

Note

The user should not modify any part of the buffer after this operation is called, until the operation completes.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>ep</th>
<th>Destination endpoint handle.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>buffer</td>
<td>Pointer to the message buffer (payload).</td>
</tr>
<tr>
<td>in</td>
<td>count</td>
<td>Number of elements to send.</td>
</tr>
<tr>
<td>in</td>
<td>param</td>
<td>Operation parameters, see ucp_request_param_t.</td>
</tr>
</tbody>
</table>
Returns

- **NULL** - The send operation was completed immediately.
- **UCS_PTR_IS_ERR(_ptr)** - The send operation failed.
- otherwise - Operation was scheduled for send and can be completed at any point in time. The request handle is returned to the application in order to track progress of the message.

Examples

```c
ucp_client_server.c.
```

### 6.7.5.7 ucp_tag_send_nb()

```c
css
ucp_status_ptr_t ucp_tag_send_nb (  
    ucp_ep_h ep,  
    const void * buffer,  
    size_t count,  
    ucp_datatype_t datatype,  
    ucp_tag_t tag,  
    ucp_send_callback_t cb)
```

This routine sends a messages that is described by the local address *buffer*, size *count*, and *datatype* object to the destination endpoint *ep*. Each message is associated with a *tag* value that is used for message matching on the *receiver*. The routine is non-blocking and therefore returns immediately, however the actual send operation may be delayed. The send operation is considered completed when it is safe to reuse the source *buffer*. If the send operation is completed immediately the routine return UCS_OK and the call-back function *cb* is **not** invoked. If the operation is **not** completed immediately and no error reported then the UCP library will schedule to invoke the call-back *cb* whenever the send operation will be completed. In other words, the completion of a message can be signaled by the return code or the call-back.

**Note**

The user should not modify any part of the *buffer* after this operation is called, until the operation completes.

Parameters

<table>
<thead>
<tr>
<th>Type</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td><em>ep</em></td>
<td>Destination endpoint handle.</td>
</tr>
<tr>
<td>in</td>
<td><em>buffer</em></td>
<td>Pointer to the message buffer (payload).</td>
</tr>
<tr>
<td>in</td>
<td><em>count</em></td>
<td>Number of elements to send</td>
</tr>
<tr>
<td>in</td>
<td><em>datatype</em></td>
<td>Datatype descriptor for the elements in the buffer.</td>
</tr>
<tr>
<td>in</td>
<td><em>tag</em></td>
<td>Message tag.</td>
</tr>
<tr>
<td>in</td>
<td><em>cb</em></td>
<td>Callback function that is invoked whenever the send operation is completed. It is important to note that the call-back is only invoked in a case when the operation cannot be completed in place.</td>
</tr>
</tbody>
</table>

Returns

- **NULL** - The send operation was completed immediately.
- **UCS_PTR_IS_ERR(_ptr)** - The send operation failed.
- otherwise - Operation was scheduled for send and can be completed at any point in time. The request handle is returned to the application in order to track progress of the message. The application is responsible for releasing the handle using *ucp_request_free()* routine.
6.7.5.8 ucp_tag_send_nbr()

```c
ucs_status_t ucp_tag_send_nbr(
    ucp_ep_h ep,
    const void *buffer,
    size_t count,
    ucp_datatype_t datatype,
    ucp_tag_t tag,
    void *req)
```

This routine provides a convenient and efficient way to implement a blocking send pattern. It also completes requests faster than ucp_tag_send_nb() because:

- it always uses uct_ep_am_bcopy() to send data up to the rendezvous threshold.
- its rendezvous threshold is higher than the one used by the ucp_tag_send_nb(). The threshold is controlled by the UCX_SEND_NBR_RNDV_THRESH environment variable.
- its request handling is simpler. There is no callback and no need to allocate and free requests. In fact request can be allocated by caller on the stack.

This routine sends a messages that is described by the local address buffer, size count, and datatype object to the destination endpoint ep. Each message is associated with a tag value that is used for message matching on the receiver.

The routine is non-blocking and therefore returns immediately, however the actual send operation may be delayed. The send operation is considered completed when it is safe to reuse the source buffer. If the send operation is completed immediately the routine returns UCS_OK.

If the operation is not completed immediately and no error reported then the UCP library will fill a user provided req and return UCS_INPROGRESS status. In order to monitor completion of the operation ucp_request_check_status() should be used.

Following pseudo code implements a blocking send function:
```c
MPI_send(...)
```
```c
{  char *request;
    ucs_status_t status;
    // allocate request on the stack
    // ucp_context_query() was used to get ucp_request_size
    request = malloc(ucp_request_size);
    // note: make sure that there is enough memory before the
    // request handle
    status = ucp_tag_send_nbr(ep, ..., request + ucp_request_size);
    if (status != UCS_INPROGRESS) {
        return status;
    }
    do {
        ucp_worker_progress(worker);
        status = ucp_request_check_status(request + ucp_request_size);
    } while (status == UCS_INPROGRESS);
    return status;
}
```

Note

The user should not modify any part of the buffer after this operation is called, until the operation completes.

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ep</code></td>
<td>Destination endpoint handle.</td>
</tr>
<tr>
<td><code>buffer</code></td>
<td>Pointer to the message buffer (payload).</td>
</tr>
<tr>
<td><code>count</code></td>
<td>Number of elements to send</td>
</tr>
<tr>
<td><code>datatype</code></td>
<td>Datatype descriptor for the elements in the buffer.</td>
</tr>
<tr>
<td><code>tag</code></td>
<td>Message tag.</td>
</tr>
<tr>
<td><code>req</code></td>
<td>Request handle allocated by the user. There should be at least UCP request size bytes of available space before the req. The size of UCP request can be obtained by ucp_context_query function.</td>
</tr>
</tbody>
</table>

© 2021 Unified Communication X (UCX). All rights reserved.
Returns

UCS_OK - The send operation was completed immediately.
UCS_INPROGRESS - The send was not completed and is in progress. ucp_request_check_status() should be used to monitor req status.
Error code as defined by ucs_status_t

6.7.5.9  ucp_tag_send_sync_nb()

```c
ucs_status_ptr_t ucp_tag_send_sync_nb {
    ucp_ep_h ep,
    const void * buffer,
    size_t count,
    ucp_datatype_t datatype,
    ucp_tag_t tag,
    ucp_send_callback_t cb }
```

Same as ucp_tag_send_nb, except the request completes only after there is a remote tag match on the message (which does not always mean the remote receive has been completed). This function never completes “in-place”, and always returns a request handle.

Note

The user should not modify any part of the buffer after this operation is called, until the operation completes. Returns UCS_ERR_UNSUPPORTED if UCP_ERR_HANDLING_MODE_PEER is enabled. This is a temporary implementation-related constraint that will be addressed in future releases.

Parameters

| in  | ep          | Destination endpoint handle. |
| in  | buffer      | Pointer to the message buffer (payload). |
| in  | count       | Number of elements to send |
| in  | datatype    | Datatype descriptor for the elements in the buffer. |
| in  | tag         | Message tag. |
| in  | cb          | Callback function that is invoked whenever the send operation is completed. |

Returns

UCS_PTR_IS_ERR(_ptr) - The send operation failed.
otherwise - Operation was scheduled for send and can be completed in any point in time. The request handle is returned to the application in order to track progress of the message. The application is responsible for releasing the handle using ucp_request_free() routine.

6.7.5.10  ucp_tag_send_nbx()

```c
ucs_status_ptr_t ucp_tag_send_nbx {
    ucp_ep_h ep,
    const void * buffer,
    size_t count,
    ucp_tag_t tag,
    const ucp_request_param_t * param }
```
This routine sends a message that is described by the local address `buffer`, size `count` object to the destination endpoint `ep`. Each message is associated with a `tag` value that is used for message matching on the `ucp_tag_recv_nb` or `receiver`. The routine is non-blocking and therefore returns immediately, however the actual send operation may be delayed. The send operation is considered completed when it is safe to reuse the source `buffer`. If the send operation is completed immediately the routine returns UCS_OK and the call-back function is not invoked. If the operation is not completed immediately and no error reported then the UCP library will schedule to invoke the call-back whenever the send operation is completed. In other words, the completion of a message can be signaled by the return code or the call-back. Immediate completion signals can be fine-tuned via the `ucp_request_param_t::op_attr_mask` field in the `ucp_request_param_t` structure. The values of this field are a bit-wise OR of the `ucp_op_attr_t` enumeration.

**Note**

The user should not modify any part of the `buffer` after this operation is called, until the operation completes.

### Parameters

| in  | ep            | Destination endpoint handle. |
| in  | buffer        | Pointer to the message buffer (payload). |
| in  | count         | Number of elements to send |
| in  | tag           | Message tag. |
| in  | param         | Operation parameters, see `ucp_request_param_t` |

### Returns

- UCS_OK - The send operation was completed immediately.
- UCS_PTR_IS_ERR(_ptr) - The send operation failed.
- otherwise - Operation was scheduled for send and can be completed in any point in time. The request handle is returned to the application in order to track progress of the message.

### Examples

ucp_client_server.c, and ucp_hello_world.c.

### 6.7.5.11 ucp_tag_send_sync_nbx()

```c
ucp_status_ptr_t ucp_tag_send_sync_nbx ( 
    ucp_ep_h ep, 
    const void * buffer, 
    size_t count, 
    ucp_tag_t tag, 
    const ucp_request_param_t * param )
```

Same as `ucp_tag_send_nbx`, except the request completes only after there is a remote tag match on the message (which does not always mean the remote receive has been completed). This function never completes "in-place", and always returns a request handle.

**Note**

The user should not modify any part of the `buffer` after this operation is called, until the operation completes. Returns UCS_ERR_UNSUPPORTED if UCP_ERR_HANDLING_MODE_PEER is enabled. This is a temporary implementation-related constraint that will be addressed in future releases.
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ep</code></td>
<td>Destination endpoint handle.</td>
</tr>
<tr>
<td><code>buffer</code></td>
<td>Pointer to the message buffer (payload).</td>
</tr>
<tr>
<td><code>count</code></td>
<td>Number of elements to send</td>
</tr>
<tr>
<td><code>tag</code></td>
<td>Message tag.</td>
</tr>
<tr>
<td><code>param</code></td>
<td>Operation parameters, see <code>ucp_request_param_t</code></td>
</tr>
</tbody>
</table>

Returns

UCS_OK - The send operation was completed immediately.
UCS_PTR_IS_ERR(_ptr) - The send operation failed.
otherwise - Operation was scheduled for send and can be completed in any point in time. The request handle is returned to the application in order to track progress of the message.

6.7.5.12  `ucp_stream_recv_nb()`

```c
ucp_status_ptr_t ucp_stream_recv_nb ( 
    ucp_ep_h ep,  
    void * buffer,  
    size_t count,  
    ucp_datatype_t datatype,  
    ucp_stream_recv_callback_t cb,  
    size_t * length,  
    unsigned flags )
```

This routine receives data that is described by the local address `buffer`, size `count`, and `datatype` object on the endpoint `ep`. The routine is non-blocking and therefore returns immediately. The receive operation is considered complete when the message is delivered to the buffer. If data is not immediately available, the operation will be scheduled for receive and a request handle will be returned. In order to notify the application about completion of a scheduled receive operation, the UCP library will invoke the call-back `cb` when data is in the receive buffer and ready for application access. If the receive operation cannot be started, the routine returns an error.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ep</code></td>
<td>UCP endpoint that is used for the receive operation.</td>
</tr>
<tr>
<td><code>buffer</code></td>
<td>Pointer to the buffer to receive the data to.</td>
</tr>
<tr>
<td><code>count</code></td>
<td>Number of elements to receive into <code>buffer</code>.</td>
</tr>
<tr>
<td><code>datatype</code></td>
<td>Datatype descriptor for the elements in the buffer.</td>
</tr>
<tr>
<td><code>cb</code></td>
<td>Callback function that is invoked whenever the receive operation is completed and the data is ready in the receive <code>buffer</code>. It is important to note that the call-back is only invoked in a case when the operation cannot be completed immediately.</td>
</tr>
<tr>
<td><code>length</code></td>
<td>Size of the received data in bytes. The value is valid only if return code is UCS_OK.</td>
</tr>
</tbody>
</table>

Note

The amount of data received, in bytes, is always an integral multiple of the `datatype` size.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>flags</code></td>
<td>Flags defined in <code>ucp_stream_recv_flags_t</code>.</td>
</tr>
</tbody>
</table>
6.7 UCP Communication routines

Returns

NULL - The receive operation was completed immediately.
UCS_PTR_IS_ERR(_ptr) - The receive operation failed.
otherwise - Operation was scheduled for receive. A request handle is returned to the application in order to track progress of the operation. The application is responsible for releasing the handle by calling the ucp_request_free routine.

6.7.5.13 ucp_stream_recv_nbx()

```c
ucs_status_ptr_t ucp_stream_recv_nbx (ucp_ep_h ep, void *buffer, size_t count, size_t *length, const ucp_request_param_t *param)
```

This routine receives data that is described by the local address `buffer`, size `count` object on the endpoint `ep`. The routine is non-blocking and therefore returns immediately. The receive operation is considered complete when the message is delivered to the buffer. If the receive operation cannot be started, the routine returns an error.

Parameters

| in | ep     | UCP endpoint that is used for the receive operation. |
| in | buffer | Pointer to the buffer that will receive the data. |
| in | count  | Number of elements to receive into `buffer`. |
| out| length | Size of the received data in bytes. The value is valid only if return code is NULL. |
| in | param  | Operation parameters, see ucp_request_param_t. This operation supports specific flags, which can be passed in `param` by ucp_request_param_t::flags. The exact set of flags is defined by ucp_stream_recv_flags_t. |

Returns

NULL - The receive operation was completed immediately. In this case the value pointed by `length` is updated by the size of received data. Note `param->recv_info` is not relevant for this function.
UCS_PTR_IS_ERR(_ptr) - The receive operation failed.
otherwise - Operation was scheduled for receive. A request handle is returned to the application in order to track progress of the operation.

Note

The amount of data received, in bytes, is always an integral multiple of the `datatype` size.

Examples

ucp_client_server.c.

6.7.5.14 ucp_stream_recv_data_nb()

```c
ucs_status_ptr_t ucp_stream_recv_data_nb (ucp_ep_h ep, size_t *length )
```
This routine receives any available data from endpoint ep. Unlike ucp_stream_recv_nb, the returned data is unstructured and is treated as an array of bytes. If data is immediately available, UCS_STATUS_PTR(_ptr) is returned as a pointer to the data, and length is set to the size of the returned data buffer. The routine is non-blocking and therefore returns immediately.

Parameters

| in   | ep             | UCP endpoint that is used for the receive operation. |
| out  | length         | Length of received data.                           |

Returns

NULL - No received data available on the ep.
UCS_PTR_IS_ERR(_ptr) - the receive operation failed and UCS_PTR_STATUS(_ptr) indicates an error.
otherwise - The pointer to the data UCS_STATUS_PTR(_ptr) is returned to the application. After the data is processed, the application is responsible for releasing the data buffer by calling the ucp_stream_data_release routine.

Note

This function returns packed data (equivalent to ucp_dt_make_contig(1)).
This function returns a pointer to a UCP-supplied buffer, whereas ucp_stream_recv_nb places the data into a user-provided buffer. In some cases, receiving data directly into a UCP-supplied buffer can be more optimal, for example by processing the incoming data in-place and thus avoiding extra memory copy operations.

6.7.5.15 ucp_tag_recv_nb()

```c
ucp_tag_recv_nb (ucp_worker_h worker, void * buffer, size_t count, ucp_datatype_t datatype, ucp_tag_t tag, ucp_tag_t tag_mask, ucp_tag_recv_callback_t cb )
```

This routine receives a message that is described by the local address buffer, size count, and datatype object on the worker. The tag value of the receive message has to match the tag and tag_mask values, where the tag_mask indicates which bits of the tag have to be matched. The routine is non-blocking and therefore returns immediately. The receive operation is considered completed when the message is delivered to the buffer. In order to notify the application about completion of the receive operation the UCP library will invoke the call-back cb when the received message is in the receive buffer and ready for application access. If the receive operation cannot be stated the routine returns an error.

Note

This routine cannot return UCS_OK. It always returns a request handle or an error.

Parameters

| in  | worker | UCP worker that is used for the receive operation. |
| in  | buffer | Pointer to the buffer to receive the data to.     |
| in  | count  | Number of elements to receive                     |
| in  | datatype | Datatype descriptor for the elements in the buffer. |
6.7 UCP Communication routines

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>tag</th>
<th>Message tag to expect.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>tag_mask</td>
<td>Bit mask that indicates the bits that are used for the matching of the incoming tag against the expected tag.</td>
</tr>
<tr>
<td>in</td>
<td>cb</td>
<td>Callback function that is invoked whenever the receive operation is completed and the data is ready in the receive buffer.</td>
</tr>
</tbody>
</table>

Returns

- UCS_PTR_IS_ERR(_ptr) - The receive operation failed.
- Otherwise - Operation was scheduled for receive. The request handle is returned to the application in order to track progress of the operation. The application is responsible for releasing the handle using ucp_request_free() routine.

6.7.5.16 ucp_tag_recv_nbr()

```c
ucs_status_t ucp_tag_recv_nbr (    ucp_worker_h worker,    void * buffer,    size_t count,    ucp_datatype_t datatype,    ucp_tag_t tag,    ucp_tag_t tag_mask,    void * req )
```

This routine receives a message that is described by the local address buffer, size count, and datatype object on the worker. The tag value of the receive message has to match the tag and tag_mask values, where the tag_mask indicates which bits of the tag have to be matched. The routine is non-blocking and therefore returns immediately. The receive operation is considered completed when the message is delivered to the buffer. In order to monitor completion of the operation ucp_request_check_status or ucp_tag_recv_request_test should be used.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>worker</th>
<th>UCP worker that is used for the receive operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>buffer</td>
<td>Pointer to the buffer to receive the data to.</td>
</tr>
<tr>
<td>in</td>
<td>count</td>
<td>Number of elements to receive.</td>
</tr>
<tr>
<td>in</td>
<td>datatype</td>
<td>Datatype descriptor for the elements in the buffer.</td>
</tr>
<tr>
<td>in</td>
<td>tag</td>
<td>Message tag to expect.</td>
</tr>
<tr>
<td>in</td>
<td>tag_mask</td>
<td>Bit mask that indicates the bits that are used for the matching of the incoming tag against the expected tag.</td>
</tr>
<tr>
<td>in</td>
<td>req</td>
<td>Request handle allocated by the user. There should be at least UCP request size bytes of available space before the req. The size of UCP request can be obtained by ucp_context_query function.</td>
</tr>
</tbody>
</table>

Returns

- Error code as defined by ucs_status_t
6.7.5.17  ucp_tag_recv_nb()  

```c
ucs_status_ptr_t ucp_tag_recv_nb (  
    ucp_worker_h worker,  
    void * buffer,  
    size_t count,  
    ucp_tag_t tag,  
    ucp_tag_t tag_mask,  
    const ucp_request_param_t * param )
```

This routine receives a message that is described by the local address `buffer`, size `count`, and `info` object on the `worker`. The tag value of the receive message has to match the `tag` and `tag_mask` values, where the `tag_mask` indicates what bits of the tag have to be matched. The routine is a non-blocking and therefore returns immediately. The receive operation is considered completed when the message is delivered to the `buffer`. In order to notify the application about completion of the receive operation the UCP library will invoke the call-back `cb` when the received message is in the receive buffer and ready for application access. If the receive operation cannot be stated the routine returns an error.

```text
Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>worker</th>
<th>UCP worker that is used for the receive operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>buffer</td>
<td>Pointer to the buffer to receive the data to.</td>
</tr>
<tr>
<td>in</td>
<td>count</td>
<td>Number of elements to receive</td>
</tr>
<tr>
<td>in</td>
<td>tag</td>
<td>Message tag to expect.</td>
</tr>
<tr>
<td>in</td>
<td>tag_mask</td>
<td>Bit mask that indicates the bits that are used for the matching of the incoming tag against the expected tag.</td>
</tr>
<tr>
<td>in</td>
<td>param</td>
<td>Operation parameters, see ucp_request_param_t</td>
</tr>
</tbody>
</table>
```

```text
Returns

NULL - The receive operation was completed immediately. In this case, if `param->recv_info.tag_info` is specified in the `param`, the value to which it points is updated with the information about the received message.

UCS_PTR_IS_ERR(_ptr) - The receive operation failed.

otherwise - Operation was scheduled for receive. The request handle is returned to the application in order to track progress of the operation. The application is responsible for releasing the handle using `ucp_request_free()` routine.
```

```text
Examples

ucp_client_server.c.
```

6.7.5.18  ucp_tag_probe_nb()  

```c
ucp_tag_message_h ucp_tag_probe_nb (  
    ucp_worker_h worker,  
    ucp_tag_t tag,  
    ucp_tag_t tag_mask,  
    int remove,  
    ucp_tag_recv_info_t * info )
```

This routine probes (checks) if a messages described by the `tag` and `tag_mask` was received (fully or partially) on the `worker`. The tag value of the received message has to match the `tag` and `tag_mask` values, where the `tag_mask` indicates what bits of the tag have to be matched. The function returns immediately and if the message is matched it returns a handle for the message.
Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>worker</th>
<th>UCP worker that is used for the probe operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>tag</td>
<td>Message tag to probe for.</td>
</tr>
<tr>
<td>in</td>
<td>tag_mask</td>
<td>Bit mask that indicates the bits that are used for the matching of the incoming tag against the expected tag.</td>
</tr>
<tr>
<td>in</td>
<td>remove</td>
<td>The flag indicates if the matched message has to be removed from UCP library. If true (1), the message handle is removed from the UCP library and the application is responsible to call ucp_tag_msg_recv_nb() in order to receive the data and release the resources associated with the message handle. If false (0), the return value is merely an indication to whether a matching message is present, and it cannot be used in any other way, and in particular it cannot be passed to ucp_tag_msg_recv_nb().</td>
</tr>
<tr>
<td>out</td>
<td>info</td>
<td>If the matching message is found the descriptor is filled with the details about the message.</td>
</tr>
</tbody>
</table>

Returns

NULL - No match found.  
Message handle (not NULL) - If message is matched the message handle is returned.

Note

This function does not advance the communication state of the network. If this routine is used in busy-poll mode, need to make sure ucp_worker_progress() is called periodically to extract messages from the transport.

Examples

ucp_hello_world.c.

6.7.5.19 ucp_tag_msg_recv_nb()

This routine receives a message that is described by the local address buffer, size count, message handle, and datatype object on the worker. The message handle can be obtained by calling the ucp_tag_probe_nb() routine. The ucp_tag_msg_recv_nb() routine is non-blocking and therefore returns immediately. The receive operation is considered completed when the message is delivered to the buffer. In order to notify the application about completion of the receive operation the UCP library will invoke the callback cb when the received message is in the receive buffer and ready for application access. If the receive operation cannot be started the routine returns an error.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>worker</th>
<th>UCP worker that is used for the receive operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>buffer</td>
<td>Pointer to the buffer that will receive the data.</td>
</tr>
<tr>
<td>in</td>
<td>count</td>
<td>Number of elements to receive</td>
</tr>
<tr>
<td>in</td>
<td>datatype</td>
<td>Datatype descriptor for the elements in the buffer.</td>
</tr>
<tr>
<td>in</td>
<td>message</td>
<td>Message handle.</td>
</tr>
<tr>
<td>in</td>
<td>cb</td>
<td>Callback function that is invoked whenever the receive operation is completed and the data is ready in the receive buffer.</td>
</tr>
</tbody>
</table>
Returns

UCS_PTR_IS_ERR(_ptr) - The receive operation failed.
otherwise - Operation was scheduled for receive. The request handle is returned to the application in
order to track progress of the operation. The application is responsible for releasing the handle using
ucp_request_free() routine.

Examples

ucp_hello_world.c.

6.7.5.20 ucp_tag_msg_recv_nbx()

ucp_tag_msg_recv_nbx (ucp_worker_h worker, void * buffer, size_t count,
ucp_tag_message_h message, const ucp_request_param_t * param )

This routine receives a message that is described by the local address buffer, size count, and message han-
dle on the worker. The message handle can be obtained by calling the ucp_tag_probe_nb() routine. The
ucp_tag_msg_recv_nbx() routine is non-blocking and therefore returns immediately. The receive operation is con-sidered completed when the message is delivered to the buffer. In order to notify the application about completion
of the receive operation the UCP library will invoke the call-back cb when the received message is in the receive
buffer and ready for application access. If the receive operation cannot be started the routine returns an error.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>worker</th>
<th>UCP worker that is used for the receive operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>buffer</td>
<td>Pointer to the buffer that will receive the data.</td>
</tr>
<tr>
<td>in</td>
<td>count</td>
<td>Number of elements to receive</td>
</tr>
<tr>
<td>in</td>
<td>message</td>
<td>Message handle.</td>
</tr>
<tr>
<td>in</td>
<td>param</td>
<td>Operation parameters, see ucp_request_param_t</td>
</tr>
</tbody>
</table>

Returns

UCS_PTR_IS_ERR(_ptr) - The receive operation failed.
otherwise - Operation was scheduled for receive. The request handle is returned to the application in
order to track progress of the operation. The application is responsible for releasing the handle using
ucp_request_free() routine.

6.7.5.21 ucp_put_nbi()

ucp_put_nbi (ucp_ep_h ep, const void * buffer, size_t length, uint64_t remote_addr,
ucp_rkey_h rkey )

This routine initiates a storage of contiguous block of data that is described by the local address buffer in the
remote contiguous memory region described by remote_addr address and the memoryhandle* rkey. The routine
returns immediately and **does not** guarantee re-usability of the source address `buffer`. If the operation is completed immediately the routine return UCS_OK, otherwise UCS_INPROGRESS or an error is returned to user.

**Note**

A user can use `ucp_worker_flush_nb()` in order to guarantee re-usability of the source address `buffer`.

**Parameters**

<table>
<thead>
<tr>
<th>in</th>
<th><code>ep</code></th>
<th>Remote endpoint handle.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td><code>buffer</code></td>
<td>Pointer to the local source address.</td>
</tr>
<tr>
<td>in</td>
<td><code>length</code></td>
<td>Length of the data (in bytes) stored under the source address.</td>
</tr>
<tr>
<td>in</td>
<td><code>remote_addr</code></td>
<td>Pointer to the destination remote memory address to write to.</td>
</tr>
<tr>
<td>in</td>
<td><code>rkey</code></td>
<td>Remote memory key associated with the remote memory address.</td>
</tr>
</tbody>
</table>

**Returns**

Error code as defined by `ucs_status_t`

### 6.7.5.22 ucp_put_nb()

```c
ucs_status_ptr_t ucp_put_nb (  
    ucp_ep_h ep,  
    const void * buffer,  
    size_t length,  
    uint64_t remote_addr,  
    ucp_rkey_h rkey,  
    ucp_send_callback_t cb )
```

This routine initiates a storage of contiguous block of data that is described by the local address `buffer` in the remote contiguous memory region described by `remote_addr` address and the `memoryhandle` `rkey`. The routine returns immediately and **does not** guarantee re-usability of the source address `buffer`. If the operation is completed immediately the routine return UCS_OK, otherwise UCS_INPROGRESS or an error is returned to user. If the put operation completes immediately, the routine returns UCS_OK and the call-back routine `cb` is **not** invoked. If the operation is **not** completed immediately and no error is reported, then the UCP library will schedule invocation of the call-back routine `cb` upon completion of the put operation. In other words, the completion of a put operation can be signaled by the return code or execution of the call-back.

**Note**

A user can use `ucp_worker_flush_nb()` in order to guarantee re-usability of the source address `buffer`.

**Parameters**

<table>
<thead>
<tr>
<th>in</th>
<th><code>ep</code></th>
<th>Remote endpoint handle.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td><code>buffer</code></td>
<td>Pointer to the local source address.</td>
</tr>
<tr>
<td>in</td>
<td><code>length</code></td>
<td>Length of the data (in bytes) stored under the source address.</td>
</tr>
<tr>
<td>in</td>
<td><code>remote_addr</code></td>
<td>Pointer to the destination remote memory address to write to.</td>
</tr>
<tr>
<td>in</td>
<td><code>rkey</code></td>
<td>Remote memory key associated with the remote memory address.</td>
</tr>
<tr>
<td>in</td>
<td><code>cb</code></td>
<td>Call-back function that is invoked whenever the put operation is completed and the local buffer can be modified. Does not guarantee remote completion.</td>
</tr>
</tbody>
</table>
Returns

NULL - The operation was completed immediately.
UCS_PTR_IS_ERR(_ptr) - The operation failed.
otherwise - Operation was scheduled and can be completed at any point in time. The request handle is returned to the application in order to track progress of the operation. The application is responsible for releasing the handle using ucp_request_free() routine.

6.7.5.23  ucp_put_nbx()

```c
ucs_status_ptr_t ucp_put_nbx (ucp_ep_h ep,
const void * buffer,
size_t count,
uint64_t remote_addr,
ucp_rkey_h rkey,
const ucp_request_param_t * param )
```

This routine initiates a storage of contiguous block of data that is described by the local address `buffer` in the remote contiguous memory region described by `remote_addr` address and the `rkey` memory handle. The routine returns immediately and does not guarantee re-usability of the source address `buffer`. If the operation is completed immediately the routine return UCS_OK, otherwise UCS_INPROGRESS or an error is returned to user. If the put operation completes immediately, the routine returns UCS_OK and the call-back routine `param.cb.send` is not invoked. If the operation is not completed immediately and no error is reported, then the UCP library will schedule invocation of the call-back routine `param.cb.send` upon completion of the put operation. In other words, the completion of a put operation can be signaled by the return code or execution of the call-back. Immediate completion signals can be fine-tuned via the `ucp_request_param_t::op_attr_mask` field in the `ucp_request_param_t` structure. The values of this field are a bit-wise OR of the `ucp_op_attr_t` enumeration.

Note

A user can use `ucp_worker_flush_nb()` in order to guarantee re-usability of the source address `buffer`.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>ep</th>
<th>Remote endpoint handle.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>buffer</td>
<td>Pointer to the local source address.</td>
</tr>
<tr>
<td>in</td>
<td>count</td>
<td>Number of elements of type <code>ucp_request_param_t::datatype</code> to put. If <code>ucp_request_param_t::datatype</code> is not specified, the type defaults to <code>ucp_dt_make_contig(1)</code>, which corresponds to byte elements.</td>
</tr>
<tr>
<td>in</td>
<td>remote_addr</td>
<td>Pointer to the destination remote memory address to write to.</td>
</tr>
<tr>
<td>in</td>
<td>rkey</td>
<td>Remote memory key associated with the remote memory address.</td>
</tr>
<tr>
<td>in</td>
<td>param</td>
<td>Operation parameters, see <code>ucp_request_param_t</code></td>
</tr>
</tbody>
</table>

Returns

UCS_OK - The operation was completed immediately.
UCS_PTR_IS_ERR(_ptr) - The operation failed.
otherwise - Operation was scheduled and can be completed at any point in time. The request handle is returned to the application in order to track progress of the operation. The application is responsible for releasing the handle using `ucp_request_free()` routine.
Note

Only the datatype \texttt{ucp_dt_make_contig(1)} is supported for \texttt{param->datatype}, see \texttt{ucp_dt_make_contig}.

6.7.5.24 \texttt{ucp_get_nbi()}

\begin{verbatim}
ucs_status_t ucp_get_nbi (  
    ucp_ep_h ep,  
    void * buffer,  
    size_t length,  
    uint64_t remote_addr,  
    ucp_rkey_h rkey )
\end{verbatim}

This routine initiate a load of contiguous block of data that is described by the remote memory address \texttt{remote_addr} and the memory handle \texttt{rkey} in the local contiguous memory region described by \texttt{buffer} address. The routine returns immediately and \textbf{does not} guarantee that remote data is loaded and stored under the local address \texttt{buffer}.

Note

A user can use \texttt{ucp_worker_flush_nb()} in order guarantee that remote data is loaded and stored under the local address \texttt{buffer}.

Parameters

\begin{itemize}
  \item \texttt{ep} Remote endpoint handle.
  \item \texttt{buffer} Pointer to the local destination address.
  \item \texttt{length} Length of the data (in bytes) stored under the destination address.
  \item \texttt{remote_addr} Pointer to the source remote memory address to read from.
  \item \texttt{rkey} Remote memory key associated with the remote memory address.
\end{itemize}

Returns

Error code as defined by \texttt{ucs_status_t}

6.7.5.25 \texttt{ucp_get_nb()}

\begin{verbatim}
ucs_status_ptr_t ucp_get_nb (  
    ucp_ep_h ep,  
    void * buffer,  
    size_t length,  
    uint64_t remote_addr,  
    ucp_rkey_h rkey,  
    ucp_send_callback_t cb )
\end{verbatim}

This routine initiates a load of a contiguous block of data that is described by the remote memory address \texttt{remote_addr} and the memory handle \texttt{rkey} in the local contiguous memory region described by \texttt{buffer} address. The routine returns immediately and \textbf{does not} guarantee that remote data is loaded and stored under the local address \texttt{buffer}. If the operation is completed immediately the routine return UCS_OK, otherwise UCS_INPROGRESS or an error is returned to user. If the get operation completes immediately, the routine returns UCS_OK and the call-back routine \texttt{cb} is \textbf{not} invoked. If the operation is \textbf{not} completed immediately and no error is reported, then the UCP library will schedule invocation of the call-back routine \texttt{cb} upon completion of the get operation. In other words, the completion of a get operation can be signaled by the return code or execution of the call-back.
Note

A user can use `ucp_worker_flush_nb()` in order to guarantee re-usability of the source address `buffer`.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ep</code></td>
<td>Remote endpoint handle.</td>
</tr>
<tr>
<td><code>buffer</code></td>
<td>Pointer to the local destination address.</td>
</tr>
<tr>
<td><code>length</code></td>
<td>Length of the data (in bytes) stored under the destination address.</td>
</tr>
<tr>
<td><code>remote_addr</code></td>
<td>Pointer to the source remote memory address to read from.</td>
</tr>
<tr>
<td><code>rkey</code></td>
<td>Remote memory key associated with the remote memory address.</td>
</tr>
<tr>
<td><code>cb</code></td>
<td>Call-back function that is invoked whenever the get operation is completed and the data is visible to the local process.</td>
</tr>
</tbody>
</table>

Returns

- NULL - The operation was completed immediately.
- UCS_PTR_IS_ERR(_ptr) - The operation failed.
- otherwise - Operation was scheduled and can be completed at any point in time. The request handle is returned to the application in order to track progress of the operation. The application is responsible for releasing the handle using `ucp_request_free()` routine.

### 6.7.5.26 ucp_get_nbx()

```c
ucp_status_ptr_t ucp_get_nbx(
    ucp_ep_h ep,
    void * buffer,
    size_t count,
    uint64_t remote_addr,
    ucp_rkey_h rkey,
    const ucp_request_param_t * param
)
```

This routine initiates a load of a contiguous block of data that is described by the remote memory address `remote_addr` and the memory handle `rkey` in the local contiguous memory region described by `buffer` address. The routine returns immediately and does not guarantee that remote data is loaded and stored under the local address `buffer`. If the operation is completed immediately the routine return UCS_OK, otherwise UCS_INPROGRESS or an error is returned to user. If the get operation completes immediately, the routine returns UCS_OK and the call-back routine `param.cb.send` is not invoked. If the operation is not completed immediately and no error is reported, then the UCP library will schedule invocation of the call-back routine `param.cb.send` upon completion of the get operation. In other words, the completion of a get operation can be signaled by the return code or execution of the call-back.

Note

A user can use `ucp_worker_flush_nb()` in order to guarantee re-usability of the source address `buffer`.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ep</code></td>
<td>Remote endpoint handle.</td>
</tr>
<tr>
<td><code>buffer</code></td>
<td>Pointer to the local destination address.</td>
</tr>
<tr>
<td><code>count</code></td>
<td>Number of elements of type <code>ucp_request_param_t::datatype</code> to put. If <code>ucp_request_param_t::datatype</code> is not specified, the type defaults to <code>ucp_dt_make_contig(1)</code>, which corresponds to byte elements.</td>
</tr>
<tr>
<td><code>remote_addr</code></td>
<td>Pointer to the source remote memory address to read from.</td>
</tr>
<tr>
<td><code>rkey</code></td>
<td>Remote memory key associated with the remote memory address.</td>
</tr>
<tr>
<td><code>param</code></td>
<td>Operation parameters, see <code>ucp_request_param_t</code>.</td>
</tr>
</tbody>
</table>
6.7 UCP Communication routines

Returns

UCS_OK - The operation was completed immediately.
UCS_PTR_IS_ERR(_ptr) - The operation failed.
otherwise - Operation was scheduled and can be completed at any point in time. The request handle is returned to the application in order to track progress of the operation. The application is responsible for releasing the handle using ucp_request_free() routine.

Note

Only the datatype ucp_dt_make_contig(1) is supported for param->datatype, see ucp_dt_make_contig.

6.7.5.27 ucp_atomic_post()

```c
ucs_status_t ucp_atomic_post (
    ucp_ep_h ep,
    ucp_atomic_post_op_t opcode,
    uint64_t value,
    size_t op_size,
    uint64_t remote_addr,
    ucp_rkey_h rkey )
```

This routine posts an atomic memory operation to a remote value. The remote value is described by the combination of the remote memory address `remote_addr` and the remote memory handle `rkey`. Return from the function does not guarantee completion. A user must call ucp_ep_flush_nb or ucp_worker_flush_nb to guarantee that the remote value has been updated.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>ep</th>
<th>UCP endpoint.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>opcode</td>
<td>One of ucp_atomic_post_op_t.</td>
</tr>
<tr>
<td>in</td>
<td>value</td>
<td>Source operand for the atomic operation.</td>
</tr>
<tr>
<td>in</td>
<td>op_size</td>
<td>Size of value in bytes</td>
</tr>
<tr>
<td>in</td>
<td>remote_addr</td>
<td>Remote address to operate on.</td>
</tr>
<tr>
<td>in</td>
<td>rkey</td>
<td>Remote key handle for the remote memory address.</td>
</tr>
</tbody>
</table>

Returns

Error code as defined by ucs_status_t

6.7.5.28 ucp_atomic_fetch_nb()

```c
ucs_status_ptr_t ucp_atomic_fetch_nb (
    ucp_ep_h ep,
    ucp_atomic_fetch_op_t opcode,
    uint64_t value,
    void * result,
    size_t op_size,
    uint64_t remote_addr,
    ucp_rkey_h rkey,
    ucp_send_callback_t cb )
```
This routine will post an atomic fetch operation to remote memory. The remote value is described by the combination of the remote memory address `remote_addr` and the remote memory handle `rkey`. The routine is non-blocking and therefore returns immediately. However, the actual atomic operation may be delayed. The atomic operation is not considered complete until the values in remote and local memory are completed. If the atomic operation completes immediately, the routine returns UCS_OK and the call-back routine `cb` is not invoked. If the operation is not completed immediately and no error is reported, then the UCP library will schedule invocation of the call-back routine `cb` upon completion of the atomic operation. In other words, the completion of an atomic operation can be signaled by the return code or execution of the call-back.

**Note**

The user should not modify any part of the `result` after this operation is called, until the operation completes.

**Parameters**

<table>
<thead>
<tr>
<th>Type</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>in</code></td>
<td><code>ep</code></td>
<td>UCP endpoint.</td>
</tr>
<tr>
<td><code>in</code></td>
<td><code>opcode</code></td>
<td>One of <code>ucp_atomic_fetch_op_t</code>.</td>
</tr>
<tr>
<td><code>in</code></td>
<td><code>value</code></td>
<td>Source operand for atomic operation. In the case of CSWAP this is the conditional for the swap. For SWAP this is the value to be placed in remote memory.</td>
</tr>
<tr>
<td><code>in,out</code></td>
<td><code>result</code></td>
<td>Local memory address to store resulting fetch to. In the case of CSWAP the value in result will be swapped into the <code>remote_addr</code> if the condition is true.</td>
</tr>
<tr>
<td><code>in</code></td>
<td><code>op_size</code></td>
<td>Size of value in bytes and pointer type for result</td>
</tr>
<tr>
<td><code>in</code></td>
<td><code>remote_addr</code></td>
<td>Remote address to operate on.</td>
</tr>
<tr>
<td><code>in</code></td>
<td><code>rkey</code></td>
<td>Remote key handle for the remote memory address.</td>
</tr>
<tr>
<td><code>in</code></td>
<td><code>cb</code></td>
<td>Call-back function that is invoked whenever the send operation is completed. It is important to note that the call-back function is only invoked in a case when the operation cannot be completed in place.</td>
</tr>
</tbody>
</table>

**Returns**

- `NULL` - The operation was completed immediately.
- `UCS_PTR_IS_ERR(_ptr)` - The operation failed.
- otherwise - Operation was scheduled and can be completed at any point in time. The request handle is returned to the application in order to track progress of the operation. The application is responsible for releasing the handle using `ucp_request_free()` routine.

### 6.7.5.29 ucp_atomic_op_nbx()

```c
ucs_status_ptr_t ucp_atomic_op_nbx ( 
    ucp_ep_h ep, 
    ucp_atomic_op_t opcode, 
    const void * buffer, 
    size_t count, 
    uint64_t remote_addr, 
    ucp_rkey_h rkey, 
    const ucp_request_param_t * param )
```

This routine will post an atomic operation to remote memory. The remote value is described by the combination of the remote memory address `remote_addr` and the remote memory handle `rkey`. The routine is non-blocking and therefore returns immediately. However, the actual atomic operation may be delayed. In order to enable fetching semantics for atomic operations user has to specify `param.reply_buffer`. Please see 6.142 below for more details.
6.7 UCP Communication routines

Note

The user should not modify any part of the buffer (or also param->reply_buffer for fetch operations), until the operation completes.
Only ucp_dt_make_config(4) and ucp_dt_make_contig(8) are supported in param->datatype, see ucp_dt_make_contig. Also, currently atomic operations can handle one element only. Thus, count argument must be set to 1.

Table 6.142: Atomic Operations Semantic

<table>
<thead>
<tr>
<th>Atomic Operation</th>
<th>Pseudo code</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCP_ATOMIC_OP_ADD</td>
<td>Result=Y; Y+=X</td>
<td>buffer</td>
<td>remote_addr</td>
<td>-</td>
<td>param.reply_buffer</td>
</tr>
<tr>
<td>UCP_ATOMIC_OP_SWAP</td>
<td>Result=Y; Y=X</td>
<td>buffer</td>
<td>remote_addr</td>
<td>-</td>
<td>param.reply_buffer</td>
</tr>
<tr>
<td>UCP_ATOMIC_OP_CSWAP</td>
<td>Result=Y; if (X==Y) then Y=Z</td>
<td>buffer</td>
<td>remote_addr</td>
<td>param.reply_buffer</td>
<td>param.reply_buffer</td>
</tr>
<tr>
<td>UCP_ATOMIC_OP_AND</td>
<td>Result=Y; Y&amp;=X</td>
<td>buffer</td>
<td>remote_addr</td>
<td>-</td>
<td>param.reply_buffer</td>
</tr>
<tr>
<td>UCP_ATOMIC_OP_OR</td>
<td>Result=Y; Y</td>
<td>=X</td>
<td>buffer</td>
<td>remote_addr</td>
<td>-</td>
</tr>
<tr>
<td>UCP_ATOMIC_OP_XOR</td>
<td>Result=Y; Y^=X</td>
<td>buffer</td>
<td>remote_addr</td>
<td>-</td>
<td>param.reply_buffer</td>
</tr>
</tbody>
</table>

Parameters

| in | ep | UCP endpoint. |
| in | opcode | One of ucp_atomic_op_t. |
| in | buffer | Address of operand for the atomic operation. See 6.142 for exact usage by different atomic operations. |
| in | count | Number of elements in buffer and result. The size of each element is specified by ucp_request_param_t::datatype |
| in | remote_addr | Remote address to operate on. |
| in | rkey | Remote key handle for the remote memory address. |
| in | param | Operation parameters, see ucp_request_param_t. |

Returns

NULL - The operation completed immediately.
UCS_PTR_IS_ERR(_ptr) - The operation failed.
otherwise - Operation was scheduled and can be completed at some time in the future. The request handle is returned to the application in order to track progress of the operation.

6.7.5.30 ucp_request_check_status()

ucp_status_t ucp_request_check_status ( void * request )

This routine checks the state of the request and returns its current status. Any value different from UCS_INPROGRESS means that request is in a completed state.
Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>request</th>
<th>Non-blocking request to check.</th>
</tr>
</thead>
</table>

Returns

- Error code as defined by `ucs_status_t`

Examples

`ucp_client_server.c`, and `ucp_hello_world.c`.

### 6.7.5.31 ucp_tag_recv_request_test()

```c
ucs_status_t ucp_tag_recv_request_test (  
    void * request,  
    ucp_tag_recv_info_t * info )
```

This routine checks the state and returns current status of the request returned from `ucp_tag_recv_nb` routine or the user allocated request for `ucp_tag_recv_nbr`. Any value different from UCS_INPROGRESS means that the request is in a completed state.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>request</th>
<th>Non-blocking request to check.</th>
</tr>
</thead>
<tbody>
<tr>
<td>out</td>
<td>info</td>
<td>It is filled with the details about the message available at the moment of calling.</td>
</tr>
</tbody>
</table>

Returns

- Error code as defined by `ucs_status_t`

### 6.7.5.32 ucp_stream_recv_request_test()

```c
ucs_status_t ucp_stream_recv_request_test (  
    void * request,  
    size_t * length_p )
```

This routine checks the state and returns current status of the request returned from `ucp_stream_recv_nb` routine. Any value different from UCS_INPROGRESS means that the request is in a completed state.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>request</th>
<th>Non-blocking request to check.</th>
</tr>
</thead>
<tbody>
<tr>
<td>out</td>
<td>length-pto</td>
<td>The size of the received data in bytes. This value is only valid if the status is UCS_OK. If valid, it is always an integral multiple of the datatype size associated with the request.</td>
</tr>
</tbody>
</table>

Returns

- Error code as defined by `ucs_status_t`
6.7.5.33 ucp_request_cancel()

void ucp_request_cancel (  
    ucp_worker_h worker,  
    void ∗ request )

Parameters

| in  | worker  | UCP worker.          |
| in  | request | Non-blocking request to cancel. |

This routine tries to cancels an outstanding communication request. After calling this routine, the request will be in completed or canceled (but not both) state regardless of the status of the target endpoint associated with the communication request. If the request is completed successfully, the send or receive completion callbacks (based on the type of the request) will be called with the status argument of the callback set to UCS_OK, and in a case it is canceled the status argument is set to UCS_ERR_CANCELED. It is important to note that in order to release the request back to the library the application is responsible for calling ucp_request_free().

6.7.5.34 ucp_stream_data_release()

void ucp_stream_data_release (  
    ucp_ep_h ep,  
    void ∗ data )

Parameters

| in  | ep      | Endpoint data received from.       |
| in  | data    | Data pointer to release, which was returned from ucp_stream_recv_data_nb. |

This routine releases internal UCP data buffer returned by ucp_stream_recv_data_nb when data is processed, the application can't use this buffer after calling this function.

6.7.5.35 ucp_request_free()

void ucp_request_free (  
    void ∗ request )

Parameters

| in  | request | Non-blocking request to release. |

This routine releases the non-blocking request back to the library, regardless of its current state. Communications operations associated with this request will make progress internally, however no further notifications or callbacks will be invoked for this request.

Examples

ucp_client_server.c.

6.7.5.36 ucp_request_alloc()

void * ucp_request_alloc (  

ucp_client_server.c.
ucp_worker_h worker }

**Parameters**

| in | worker | UCP worker |

**Returns**

Error code as defined by ucs_status_t

This routine creates request which may be used in functions ucp_tag_send_nb, ucp_tag_recv_nb, etc. The application is responsible for releasing the handle using the ucp_request_free routine.

6.7.5.37 ucp_request_is_completed()

```c
int ucp_request_is_completed {
    void * request }
```

**Deprecated** Replaced by ucp_request_test.

6.7.5.38 ucp_put()

```c
ucs_status_t ucp_put (  
    ucp_ep_h ep,  
    const void * buffer,  
    size_t length,  
    uint64_t remote_addr,  
    ucp_rkey_h rkey )
```

**Deprecated** Replaced by ucp_put_nb. The following example implements the same functionality using ucp_put_nb:

```c
void empty_callback(void *request, ucs_status_t status)  
{  
}  
void simple_put(ucp_ep_h ep, const void *buffer, size_t length,  
    uint64_t remote_addr, ucp_rkey_h rkey)  
{  
    void *request = ucp_put_nb(ep, buffer, length, remote_addr, rkey,  
        empty_callback),  
    if (request == NULL) {  
        return UCS_OK;  
    } else if (UCS_PTR_IS_ERR(request)) {  
        return UCS_PTR_STATUS(request);  
    } else {  
        ucs_status_t status;  
        do {  
            ucp_worker_progress(worker);  
            status = ucp_request_check_status(request);  
        } while (status == UCS_INPROGRESS);  
        ucp_request_release(request);  
        return status;  
    }  
}
```

This routine stores contiguous block of data that is described by the local address buffer in the remote contiguous memory region described by remote_addr address and the memory handle rkey. The routine returns when it is safe to reuse the source address buffer.

**Parameters**

| in | ep | Remote endpoint handle. |
6.7 UCP Communication routines

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>buffer</th>
<th>Pointer to the local source address.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>length</td>
<td>Length of the data (in bytes) stored under the source address.</td>
</tr>
<tr>
<td>in</td>
<td>remote_addr</td>
<td>Pointer to the destination remote address to write to.</td>
</tr>
<tr>
<td>in</td>
<td>rkey</td>
<td>Remote memory key associated with the remote address.</td>
</tr>
</tbody>
</table>

Returns

Error code as defined by ucs_status_t

6.7.5.39 ucp_get()

ucps_status_t ucp_get(
    ucp_ep_h ep,
    void * buffer,
    size_t length,
    uint64_t remote_addr,
    ucp_rkey_h rkey )

Deprecated Replaced by ucp_get_nb.

See also
ucp_put.

This routine loads contiguous block of data that is described by the remote address remote_addr and the memory handle rkey in the local contiguous memory region described by buffer address. The routine returns when remote data is loaded and stored under the local address buffer.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>ep</th>
<th>Remote endpoint handle.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>buffer</td>
<td>Pointer to the local source address.</td>
</tr>
<tr>
<td>in</td>
<td>length</td>
<td>Length of the data (in bytes) stored under the source address.</td>
</tr>
<tr>
<td>in</td>
<td>remote_addr</td>
<td>Pointer to the destination remote address to write to.</td>
</tr>
<tr>
<td>in</td>
<td>rkey</td>
<td>Remote memory key associated with the remote address.</td>
</tr>
</tbody>
</table>

Returns

Error code as defined by ucs_status_t

6.7.5.40 ucp_atomic_add32()

ucps_status_t ucp_atomic_add32(
    ucp_ep_h ep,
    uint32_t add,
    uint64_t remote_addr,
    ucp_rkey_h rkey )
**Deprecated** Replaced by `ucp_atomic_post` with opcode UCP_ATOMIC_POST_OP_ADD.

See also

`ucp_put`.

This routine performs an add operation on a 32 bit integer value atomically. The remote integer value is described by the combination of the remote memory address `remote_addr` and the remote memory handle `rkey`. The `add` value is the value that is used for the add operation. When the operation completes the sum of the original remote value and the operand value (`add`) is stored in remote memory. The call to the routine returns immediately, independent of operation completion.

Note

The remote address must be aligned to 32 bit.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ep</code></td>
<td>Remote endpoint handle.</td>
</tr>
<tr>
<td><code>add</code></td>
<td>Value to add.</td>
</tr>
<tr>
<td><code>remote_addr</code></td>
<td>Pointer to the destination remote address of the atomic variable.</td>
</tr>
<tr>
<td><code>rkey</code></td>
<td>Remote memory key associated with the remote address.</td>
</tr>
</tbody>
</table>

**Returns**

Error code as defined by `ucs_status_t`

---

6.7.5.41 `ucp_atomic_add64()`

```c
ucs_status_t ucp_atomic_add64 (  
    ucp_ep_h ep,  
    uint64_t add,  
    uint64_t remote_addr,  
    ucp_rkey_h rkey )
```

**Deprecated** Replaced by `ucp_atomic_post` with opcode UCP_ATOMIC_POST_OP_ADD.

See also

`ucp_put`.

This routine performs an add operation on a 64 bit integer value atomically. The remote integer value is described by the combination of the remote memory address `remote_addr` and the remote memory handle `rkey`. The `add` value is the value that is used for the add operation. When the operation completes the sum of the original remote value and the operand value (`add`) is stored in remote memory. The call to the routine returns immediately, independent of operation completion.

Note

The remote address must be aligned to 64 bit.
6.7 UCP Communication routines

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ep</code></td>
<td>Remote endpoint handle.</td>
</tr>
<tr>
<td><code>add</code></td>
<td>Value to add.</td>
</tr>
<tr>
<td><code>remote_addr</code></td>
<td>Pointer to the destination remote address of the atomic variable.</td>
</tr>
<tr>
<td><code>rkey</code></td>
<td>Remote memory key associated with the remote address.</td>
</tr>
</tbody>
</table>

Returns

Error code as defined by `ucs_status_t`

6.7.5.42 ucp_atomic_fadd32()

```c
ucs_status_t ucp_atomic_fadd32 (  
    ucp_ep_h ep,  
    uint32_t add,  
    uint64_t remote_addr,  
    ucp_rkey_h rkey,  
    uint32_t * result)  
```

Deprecated

Replaced by `ucp_atomic_fetch_nb` with opcode UCP_ATOMIC_FETCH_OP_FADD.

See also

- `ucp_put`

This routine performs an add operation on a 32 bit integer value atomically. The remote integer value is described by the combination of the remote memory address `remote_addr` and the remote memory handle `rkey`. The `add` value is the value that is used for the add operation. When the operation completes, the original remote value is stored in the local memory `result`, and the sum of the original remote value and the operand value is stored in remote memory. The call to the routine returns when the operation is completed and the `result` value is updated.

Note

The remote address must be aligned to 32 bit.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ep</code></td>
<td>Remote endpoint handle.</td>
</tr>
<tr>
<td><code>add</code></td>
<td>Value to add.</td>
</tr>
<tr>
<td><code>remote_addr</code></td>
<td>Pointer to the destination remote address of the atomic variable.</td>
</tr>
<tr>
<td><code>rkey</code></td>
<td>Remote memory key associated with the remote address.</td>
</tr>
<tr>
<td><code>result</code></td>
<td>Pointer to the address that is used to store the previous value of the atomic variable described by the <code>remote_addr</code>.</td>
</tr>
</tbody>
</table>

Returns

Error code as defined by `ucs_status_t`
6.7.5.43 ucp_atomic_fadd64()

```c
ucs_status_t ucp_atomic_fadd64 (
    ucp_ep_h ep,
    uint64_t add,
    uint64_t remote_addr,
    ucp_rkey_h rkey,
    uint64_t ∗ result )
```

**Deprecated** Replaced by `ucp_atomic_fetch_nb` with opcode UCP_ATOMIC_FETCH_OP_FADD.

See also

- `ucp_put`.

This routine performs an add operation on a 64 bit integer value atomically. The remote integer value is described by the combination of the remote memory address `remote_addr` and the remote memory handle `rkey`. The `add` value is the value that is used for the add operation. When the operation completes, the original remote value is stored in the local memory `result`, and the sum of the original remote value and the operand value is stored in remote memory. The call to the routine returns when the operation is completed and the `result` value is updated.

**Note**

The remote address must be aligned to 64 bit.

### Parameters

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td><code>ep</code></td>
<td>Remote endpoint handle.</td>
</tr>
<tr>
<td>in</td>
<td><code>add</code></td>
<td>Value to add.</td>
</tr>
<tr>
<td>in</td>
<td><code>remote_addr</code></td>
<td>Pointer to the destination remote address of the atomic variable.</td>
</tr>
<tr>
<td>in</td>
<td><code>rkey</code></td>
<td>Remote memory key associated with the remote address.</td>
</tr>
<tr>
<td>out</td>
<td><code>result</code></td>
<td>Pointer to the address that is used to store the previous value of the atomic variable described by the <code>remote_addr</code>.</td>
</tr>
</tbody>
</table>

### Returns

Error code as defined by `ucs_status_t`.

6.7.5.44 ucp_atomic_swap32()

```c
ucs_status_t ucp_atomic_swap32 (
    ucp_ep_h ep,
    uint32_t swap,
    uint64_t remote_addr,
    ucp_rkey_h rkey,
    uint32_t ∗ result )
```

**Deprecated** Replaced by `ucp_atomic_fetch_nb` with opcode UCP_ATOMIC_FETCH_OP_SWAP.
6.7 UCP Communication routines

See also
ucp_put.

This routine swaps a 32 bit value between local and remote memory. The remote value is described by the combination of the remote memory address `remote_addr` and the remote memory handle `rkey`. The `swap` value is the value that is used for the swap operation. When the operation completes, the remote value is stored in the local memory `result`, and the operand value (`swap`) is stored in remote memory. The call to the routine returns when the operation is completed and the `result` value is updated.

Note
The remote address must be aligned to 32 bit.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ep</code></td>
<td>Remote endpoint handle.</td>
</tr>
<tr>
<td><code>swap</code></td>
<td>Value to swap.</td>
</tr>
<tr>
<td><code>remote_addr</code></td>
<td>Pointer to the destination remote address of the atomic variable.</td>
</tr>
<tr>
<td><code>rkey</code></td>
<td>Remote memory key associated with the remote address.</td>
</tr>
<tr>
<td><code>result</code></td>
<td>Pointer to the address that is used to store the previous value of the atomic variable described by the <code>remote_addr</code></td>
</tr>
</tbody>
</table>

Returns
Error code as defined by `ucs_status_t`

6.7.5.45 ucp_atomic_swap64()

```c
ucs_status_t ucp_atomic_swap64 (  
    ucp_ep_h ep,  
    uint64_t swap,  
    uint64_t remote_addr,  
    ucp_rkey_h rkey,  
    uint64_t * result )
```

Deprecated Replaced by `ucp_atomic_fetch_nb` with opcode UCP_ATOMIC_FETCH_OP_SWAP.

See also
ucp_put.

This routine swaps a 64 bit value between local and remote memory. The remote value is described by the combination of the remote memory address `remote_addr` and the remote memory handle `rkey`. The `swap` value is the value that is used for the swap operation. When the operation completes, the remote value is stored in the local memory `result`, and the operand value (`swap`) is stored in remote memory. The call to the routine returns when the operation is completed and the `result` value is updated.

Note
The remote address must be aligned to 64 bit.
Parameters

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>ep</td>
<td>Remote endpoint handle.</td>
</tr>
<tr>
<td>in</td>
<td>swap</td>
<td>Value to swap.</td>
</tr>
<tr>
<td>in</td>
<td>remote_addr</td>
<td>Pointer to the destination remote address of the atomic variable.</td>
</tr>
<tr>
<td>in</td>
<td>rkey</td>
<td>Remote memory key associated with the remote address.</td>
</tr>
<tr>
<td>out</td>
<td>result</td>
<td>Pointer to the address that is used to store the previous value of the atomic variable described by the remote_addr</td>
</tr>
</tbody>
</table>

Returns

Error code as defined by ucs_status_t

6.7.5.46 ucp_atomic_cswap32()

ucp_status_t ucp_atomic_cswap32(
    ucp_ep_h ep,
    uint32_t compare,
    uint32_t swap,
    uint64_t remote_addr,
    ucp_rkey_h rkey,
    uint32_t * result)

Deprecated Replaced by ucp_atomic_fetch_nb with opcode UCP_ATOMIC_FETCH_OP_CSWAP.

See also

ucp_put.

This routine conditionally swaps a 32 bit value between local and remote memory. The swap occurs only if the condition value (continue) is equal to the remote value, otherwise the remote memory is not modified. The remote value is described by the combination of the remote memory address remote_addr and the remote memory handle rkey. The swap value is the value that is used to update the remote memory if the condition is true. The call to the routine returns when the operation is completed and the result value is updated.

Note

The remote address must be aligned to 32 bit.

Parameters

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>ep</td>
<td>Remote endpoint handle.</td>
</tr>
<tr>
<td>in</td>
<td>compare</td>
<td>Value to compare to.</td>
</tr>
<tr>
<td>in</td>
<td>swap</td>
<td>Value to swap.</td>
</tr>
<tr>
<td>in</td>
<td>remote_addr</td>
<td>Pointer to the destination remote address of the atomic variable.</td>
</tr>
<tr>
<td>in</td>
<td>rkey</td>
<td>Remote memory key associated with the remote address.</td>
</tr>
<tr>
<td>out</td>
<td>result</td>
<td>Pointer to the address that is used to store the previous value of the atomic variable described by the remote_addr</td>
</tr>
</tbody>
</table>
6.7 UCP Communication routines

Returns

Error code as defined by ucs_status_t

6.7.5.47 ucp_atomic_cswap64()

```c
ucp_status_t ucp_atomic_cswap64(
    ucp_ep_h ep,
    uint64_t compare,
    uint64_t swap,
    uint64_t remote_addr,
    ucp_rkey_h rkey,
    uint64_t *result)
```

Deprecated Replaced by ucp_atomic_fetch_nb with opcode UCP_ATOMIC_FETCH_OP_CSWAP.

See also

ucp_put.

This routine conditionally swaps a 64 bit value between local and remote memory. The swap occurs only if the condition value (continue) is equal to the remote value, otherwise the remote memory is not modified. The remote value is described by the combination of the remote memory address remote_addr and the remote memory handle rkey. The swap value is the value that is used to update the remote memory if the condition is true. The call to the routine returns when the operation is completed and the result value is updated.

Note

The remote address must be aligned to 64 bit.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>ep</th>
<th>Remote endpoint handle.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>compare</td>
<td>Value to compare to.</td>
</tr>
<tr>
<td>in</td>
<td>swap</td>
<td>Value to swap.</td>
</tr>
<tr>
<td>in</td>
<td>remote_addr</td>
<td>Pointer to the destination remote address of the atomic variable.</td>
</tr>
<tr>
<td>in</td>
<td>rkey</td>
<td>Remote memory key associated with the remote address.</td>
</tr>
<tr>
<td>out</td>
<td>result</td>
<td>Pointer to the address that is used to store the previous value of the atomic variable described by the remote_addr</td>
</tr>
</tbody>
</table>

Returns

Error code as defined by ucs_status_t
6.8 UCP Configuration

Data Structures

- struct ucp_params
  
  Tuning parameters for UCP library. More...

Typedefs

- typedef struct ucp_params ucp_params_t
  
  Tuning parameters for UCP library.

- typedef struct ucp_config ucp_config_t
  
  UCP configuration descriptor.

Functions

- ucs_status_t ucp_config_read (const char *env_prefix, const char *filename, ucp_config_t **config_p)
  
  Read UCP configuration descriptor.

- void ucp_config_release (ucp_config_t *config)
  
  Release configuration descriptor.

- ucs_status_t ucp_config_modify (ucp_config_t *config, const char *name, const char *value)
  
  Modify context configuration.

- void ucp_config_print (const ucp_config_t *config, FILE *stream, const char *title, ucs_config_print_flags_t print_flags)
  
  Print configuration information.

6.8.1 Detailed Description

This section describes routines for configuration of the UCP network layer.

6.8.2 Data Structure Documentation

6.8.2.1 struct ucp_params

The structure defines the parameters that are used for UCP library tuning during UCP library initialization.

Note

UCP library implementation uses the features parameter to optimize the library functionality that minimize memory footprint. For example, if the application does not require send/receive semantics UCP library may avoid allocation of expensive resources associated with send/receive queues.

Examples

ucp_client_server.c, and ucp_hello_world.c.

Data Fields

<table>
<thead>
<tr>
<th>data_type</th>
<th>field_name</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint64_t</td>
<td>field_mask</td>
<td>Mask of valid fields in this structure, using bits from ucp_params_field. Fields not specified in this mask will be ignored. Provides ABI compatibility with respect to adding new fields.</td>
</tr>
</tbody>
</table>
### Data Fields

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint64_t</td>
<td>features</td>
<td>UCP <strong>features</strong> that are used for library initialization. It is recommended for applications only to request the features that are required for an optimal functionality. This field must be specified.</td>
</tr>
<tr>
<td>size_t</td>
<td>request_size</td>
<td>The size of a reserved space in a non-blocking requests. Typically applications use this space for caching own structures in order to avoid costly memory allocations, pointer dereferences, and cache misses. For example, MPI implementation can use this memory for caching MPI descriptors. This field defaults to 0 if not specified.</td>
</tr>
<tr>
<td>ucp_request_init_callback_t</td>
<td>request_init</td>
<td>Pointer to a routine that is used for the request initialization. This function will be called only on the very first time a request memory is initialized, and may not be called again if a request is reused. If a request should be reset before the next reuse, it can be done before calling ucp_request_free. NULL can be used if no such is function required, which is also the default if this field is not specified by field_mask.</td>
</tr>
<tr>
<td>ucp_request_cleanup_callback_t</td>
<td>request_cleanup</td>
<td>Pointer to a routine that is responsible for final cleanup of the memory associated with the request. This routine may not be called every time a request is released. For some implementations, the cleanup call may be delayed and only invoked at ucp_worker_destroy. NULL can be used if no such function is required, which is also the default if this field is not specified by field_mask.</td>
</tr>
<tr>
<td>uint64_t</td>
<td>tag_sender_mask</td>
<td>Mask which specifies particular bits of the tag which can uniquely identify the sender (UCP endpoint) in tagged operations. This field defaults to 0 if not specified.</td>
</tr>
<tr>
<td>int</td>
<td>mt_workers_shared</td>
<td>This flag indicates if this context is shared by multiple workers from different threads. If so, this context needs thread safety support; otherwise, the context does not need to provide thread safety. For example, if the context is used by single worker, and that worker is shared by multiple threads, this context does not need thread safety; if the context is used by worker 1 and worker 2, and worker 1 is used by thread 1 and worker 2 is used by thread 2, then this context needs thread safety. Note that actual thread mode may be different from mode passed to ucp_init. To get actual thread mode use ucp_context_query.</td>
</tr>
</tbody>
</table>
**Data Fields**

| size_t | estimated_num_eps | An optimization hint of how many endpoints will be created on this context. For example, when used from MPI or SHMEM libraries, this number will specify the number of ranks (or processing elements) in the job. Does not affect semantics, but only transport selection criteria and the resulting performance. The value can be also set by UCX_NUM_EPS environment variable. In such case it will override the number of endpoints set by estimated_num_eps |
| size_t | estimated_num_ppn | An optimization hint for a single node. For example, when used from MPI or OpenSHMEM libraries, this number will specify the number of Processes Per Node (PPN) in the job. Does not affect semantics, only transport selection criteria and the resulting performance. The value can be also set by the UCX_NUM_PPN environment variable, which will override the number of endpoints set by estimated_num_ppn |

**6.8.3 Typedef Documentation**

**6.8.3.1 ucp_params_t**

typedef struct ucp_params ucp_params_t

The structure defines the parameters that are used for UCP library tuning during UCP library initialization.

Note

UCP library implementation uses the features parameter to optimize the library functionality that minimize memory footprint. For example, if the application does not require send/receive semantics UCP library may avoid allocation of expensive resources associated with send/receive queues.

**6.8.3.2 ucp_config_t**

typedef struct ucp_config ucp_config_t

This descriptor defines the configuration for UCP application context. The configuration is loaded from the run-time environment (using configuration files of environment variables) using ucp_config_read routine and can be printed using ucp_config_print routine. In addition, application is responsible to release the descriptor using ucp_config_release routine.

**6.8.4 Function Documentation**
6.8 UCP Configuration

6.8.4.1 ucp_config_read()

```c
ucp_status_t ucp_config_read {
    const char * env_prefix,
    const char * filename,
    ucp_config_t ** config_p
}
```

The routine fetches the information about UCP library configuration from the run-time environment. Then, the fetched descriptor is used for UCP library initialization. The Application can print out the descriptor using print routine. In addition the application is responsible for releasing the descriptor back to the UCP library.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>env_prefix</th>
<th>If non-NULL, the routine searches for the environment variables that start with <code>&lt;env_prefix&gt;_UCX</code> prefix. Otherwise, the routine searches for the environment variables that start with <code>UCX</code> prefix.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>filename</td>
<td>If non-NULL, read configuration from the file defined by <code>filename</code>. If the file does not exist, it will be ignored and no error reported to the application.</td>
</tr>
<tr>
<td>out</td>
<td>config_p</td>
<td>Pointer to configuration descriptor as defined by <code>ucp_config_t</code>.</td>
</tr>
</tbody>
</table>

Returns

Error code as defined by `ucp_status_t`

Examples

ucp_hello_world.c.

6.8.4.2 ucp_config_release()

```c
void ucp_config_release {
    ucp_config_t * config
}
```

The routine releases the configuration descriptor that was allocated through `ucp_config_read()` routine.

Parameters

| out | config | Configuration descriptor as defined by `ucp_config_t`. |

Examples

ucp_hello_world.c.

6.8.4.3 ucp_config_modify()

```c
ucp_status_t ucp_config_modify {
    ucp_config_t * config,
    const char * name,
    const char * value
}
```

The routine changes one configuration setting stored in configuration descriptor.
Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>config</th>
<th>Configuration to modify.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>name</td>
<td>Configuration variable name.</td>
</tr>
<tr>
<td>in</td>
<td>value</td>
<td>Value to set.</td>
</tr>
</tbody>
</table>

Returns

Error code.

6.8.4.4 ucp_config_print()

```c
void ucp_config_print (
    const ucp_config_t * config,
    FILE * stream,
    const char * title,
    ucs_config_print_flags_t print_flags )
```

The routine prints the configuration information that is stored in configuration descriptor.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>config</th>
<th>Configuration descriptor to print.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>stream</td>
<td>Output stream to print the configuration to.</td>
</tr>
<tr>
<td>in</td>
<td>title</td>
<td>Configuration title to print.</td>
</tr>
<tr>
<td>in</td>
<td>print_flags</td>
<td>Flags that control various printing options.</td>
</tr>
</tbody>
</table>

Examples

ucp_hello_world.c.
6.9 UCP Data type routines

Data Structures

- struct ucp_dt_iov
  
  Structure for scatter-gather I/O. More...

- struct ucp_generic_dt_ops
  
  UCP generic data type descriptor.

Macros

- #define ucp_dt_make_contig(_elem_size) (((ucp_datatype_t)(_elem_size) << UCP_DATATYPE_SHIFT) | UCP_DATATYPE_CONTIG)
  
  Generate an identifier for contiguous data type.

- #define ucp_dt_make_iov() ((ucp_datatype_t)UCP_DATATYPE_IOV)
  
  Generate an identifier for Scatter-gather IOV data type.

Typedefs

- typedef struct ucp_dt_iov ucp_dt_iov_t
  
  Structure for scatter-gather I/O.

- typedef struct ucp_generic_dt_ops ucp_generic_dt_ops_t
  
  UCP generic data type descriptor.

Enumerations

- enum ucp_dt_type {
  
  UCP_DATATYPE_CONTIG = 0, UCP_DATATYPE_STRIDED = 1, UCP_DATATYPE_IOV = 2, UCP_DATATYPE_GENERIC = 7, UCP_DATATYPE_SHIFT = 3, UCP_DATATYPE_CLASS_MASK = UCS_MASK(UCP_DATATYPE_SHIFT)
  
  UCP data type classification.

Functions

- ucs_status_t ucp_dt_create_generic (const ucp_generic_dt_ops_t *ops, void *context, ucp_datatype_t *datatype_p)
  
  Create a generic datatype.

- void ucp_dt_destroy (ucp_datatype_t datatype)
  
  Destroy a datatype and release its resources.

Variables

- void *(* ucp_generic_dt_ops::start_pack ) (void *context, const void *buffer, size_t count)
  
  Start a packing request.

- void *(* ucp_generic_dt_ops::start_unpack ) (void *context, void *buffer, size_t count)
  
  Start an unpacking request.

- size_t *(* ucp_generic_dt_ops::packed_size ) (void *state)
  
  Get the total size of packed data.

- size_t *(* ucp_generic_dt_ops::pack ) (void *state, size_t offset, void *dest, size_t max_length)
Pack data.
• `ucs_status_t(+ ucp_generic_dt_ops::unpack ) (void *state, size_t offset, const void *src, size_t length)`

Unpack data.
• `void(+ ucp_generic_dt_ops::finish ) (void *state)`

Finish packing/unpacking.

6.9.1 Detailed Description

UCP Data type routines

6.9.2 Data Structure Documentation

6.9.2.1 `struct ucp_dt_iov`

This structure is used to specify a list of buffers which can be used within a single data transfer function call.

Note

If `length` is zero, the memory pointed to by `buffer` will not be accessed. Otherwise, `buffer` must point to valid memory.

Data Fields

<table>
<thead>
<tr>
<th>void * buffer</th>
<th>Pointer to a data buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>size_t length</td>
<td>Length of the <code>buffer</code> in bytes</td>
</tr>
</tbody>
</table>

6.9.3 Macro Definition Documentation

6.9.3.1 `ucp_dt_make_contig`

```c
#define ucp_dt_make_contig(_elem_size) (((ucp_datatype_t)(_elem_size)<<UCP_DATATYPE_SHIFT) | UCP_DATATYPE_CONTIG)
```

This macro creates an identifier for contiguous datatype that is defined by the size of the basic element.

Parameters

| in _elem_size | Size of the basic element of the type. |

Returns

Data-type identifier.

Note

In case of partial receive, the buffer will be filled with integral count of elements.

Examples

ucp_hello_world.c.
6.9.3.2 `ucp_dt_make_iov`

```c
#define ucp_dt_make_iov() ((ucp_datatype_t)UCP_DATATYPE_IOV)
```

This macro creates an identifier for datatype of scatter-gather list with multiple pointers.

**Returns**

Data-type identifier.

**Note**

In the event of partial receive, `ucp_dt_iov_t::buffer` can be filled with any number of bytes according to its `ucp_dt_iov_t::length`.

### 6.9.4 Typedef Documentation

#### 6.9.4.1 `ucp_dt_iov_t`

```c
typedef struct ucp_dt_iov ucp_dt_iov_t
```

This structure is used to specify a list of buffers which can be used within a single data transfer function call.

**Note**

If `length` is zero, the memory pointed to by `buffer` will not be accessed. Otherwise, `buffer` must point to valid memory.

#### 6.9.4.2 `ucp_generic_dt_ops_t`

```c
typedef struct ucp_generic_dt_ops ucp_generic_dt_ops_t
```

This structure provides a generic datatype descriptor that is used for definition of application defined datatypes. Typically, the descriptor is used for an integration with datatype engines implemented within MPI and SHMEM implementations.

**Note**

In case of partial receive, any amount of received data is acceptable which matches buffer size.

### 6.9.5 Enumeration Type Documentation

#### 6.9.5.1 `ucp_dt_type`

```c
typedef enum ucp_dt_type
```

The enumeration list describes the datatypes supported by UCP.
**Module Documentation**

### Enumerator

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCP_DATATYPE_CONTIG</td>
<td>Contiguous datatype</td>
</tr>
<tr>
<td>UCP_DATATYPE_STRIDED</td>
<td>Strided datatype</td>
</tr>
<tr>
<td>UCP_DATATYPE_IOV</td>
<td>Scatter-gather list with multiple pointers</td>
</tr>
<tr>
<td>UCP_DATATYPE_GENERIC</td>
<td>Generic datatype with user-defined pack/unpack routines</td>
</tr>
<tr>
<td>UCP_DATATYPE_SHIFT</td>
<td>Number of bits defining the datatype classification</td>
</tr>
<tr>
<td>UCP_DATATYPE_CLASS_MASK</td>
<td>Data-type class mask</td>
</tr>
</tbody>
</table>

### 6.9.6 Function Documentation

#### 6.9.6.1 ucp_dt_create_generic()

```c
ucs_status_t ucp_dt_create_generic(
    const ucp_generic_dt_ops_t * ops,
    void * context,
    ucp_datatype_t * datatype_p)
```

This routine creates a generic datatype object. The generic datatype is described by the `ops` object which provides a table of routines defining the operations for generic datatype manipulation. Typically, generic datatypes are used for integration with datatype engines provided with MPI implementations (MPICH, Open MPI, etc). The application is responsible for releasing the `datatype_p` object using `ucp_dt_destroy()` routine.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in ops</td>
<td>Generic datatype function table as defined by <code>ucp_generic_dt_ops_t</code>.</td>
</tr>
<tr>
<td>in context</td>
<td>Application defined context passed to this routine. The context is passed as a parameter to the routines in the <code>ops</code> table.</td>
</tr>
<tr>
<td>out datatype_p</td>
<td>A pointer to datatype object.</td>
</tr>
</tbody>
</table>

**Returns**

Error code as defined by `ucs_status_t`

#### 6.9.6.2 ucp_dt_destroy()

```c
void ucp_dt_destroy(
    ucp_datatype_t datatype)
```

This routine destroys the `datatype` object and releases any resources that are associated with the object. The `datatype` object must be allocated using `ucp_dt_create_generic()` routine.

**Warning**

- Once the `datatype` object is released an access to this object may cause an undefined failure.
6.9 UCP Data type routines

Parameters

| in | datatype | Datatype object to destroy. |

6.9.7 Variable Documentation

6.9.7.1 start_pack

void*(* ucp_generic_dt_ops::start_pack) (void *context, const void *buffer, size_t count)

The pointer refers to application defined start-to-pack routine. It will be called from the ucp_tag_send_nb routine.

Parameters

| in | context | User-defined context. |
| in | buffer  | Buffer to pack. |
| in | count   | Number of elements to pack into the buffer. |

Returns

A custom state that is passed to the following pack() routine.

6.9.7.2 start_unpack

void*(* ucp_generic_dt_ops::start_unpack) (void *context, void *buffer, size_t count)

The pointer refers to application defined start-to-unpack routine. It will be called from the ucp_tag_recv_nb routine.

Parameters

| in | context | User-defined context. |
| in | buffer  | Buffer to unpack to. |
| in | count   | Number of elements to unpack in the buffer. |

Returns

A custom state that is passed later to the following unpack() routine.

6.9.7.3 packed_size

size_t(* ucp_generic_dt_ops::packed_size) (void *state)

The pointer refers to user defined routine that returns the size of data in a packed format.

Parameters

| in | state   | State as returned by start_pack() routine. |
Returns

The size of the data in a packed form.

6.9.7.4 pack

\texttt{size\_t(* ucp\_generic\_dt\_ops::pack) (void *state, size\_t offset, void *dest, size\_t max\_length)}

The pointer refers to application defined pack routine.

Parameters

<table>
<thead>
<tr>
<th>parameter</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in \textit{state}</td>
<td>State as returned by \texttt{start_pack()} routine.</td>
</tr>
<tr>
<td>in \textit{offset}</td>
<td>Virtual offset in the output stream.</td>
</tr>
<tr>
<td>in \textit{dest}</td>
<td>Destination to pack the data to.</td>
</tr>
<tr>
<td>in \textit{max_length}</td>
<td>Maximal length to pack.</td>
</tr>
</tbody>
</table>

Returns

The size of the data that was written to the destination buffer. Must be less than or equal to \textit{max\_length}.

6.9.7.5 unpack

\texttt{ucs\_status\_t(* ucp\_generic\_dt\_ops::unpack) (void *state, size\_t offset, const void *src, size\_t length)}

The pointer refers to application defined unpack routine.

Parameters

<table>
<thead>
<tr>
<th>parameter</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in \textit{state}</td>
<td>State as returned by \texttt{start_unpack()} routine.</td>
</tr>
<tr>
<td>in \textit{offset}</td>
<td>Virtual offset in the input stream.</td>
</tr>
<tr>
<td>in \textit{src}</td>
<td>Source to unpack the data from.</td>
</tr>
<tr>
<td>in \textit{length}</td>
<td>Length to unpack.</td>
</tr>
</tbody>
</table>

Returns

\texttt{UCS\_OK} or an error if unpacking failed.

6.9.7.6 finish

\texttt{void(* ucp\_generic\_dt\_ops::finish) (void *state)}

The pointer refers to application defined finish routine.

Parameters

<table>
<thead>
<tr>
<th>parameter</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in \textit{state}</td>
<td>State as returned by \texttt{start_pack()} and \texttt{start_unpack()} routines.</td>
</tr>
</tbody>
</table>
6.10 Unified Communication Transport (UCT) API

Modules

- UCT Communication Resource
- UCT Communication Context
- UCT Memory Domain
- UCT Active messages
- UCT Remote memory access operations
- UCT Atomic operations
- UCT Tag matching operations
- UCT client-server operations

6.10.1 Detailed Description

This section describes UCT API.
6.11 UCT Communication Resource

Modules

- **UCT interface operations and capabilities**
  
  *List of capabilities supported by UCX API.*

- **UCT interface for asynchronous event capabilities**
  
  *List of capabilities supported by UCT iface event API.*

Data Structures

- `struct uct_md_resource_desc`
  
  *Memory domain resource descriptor.*

- `struct uct_component_attr`
  
  *UCT component attributes.*

- `struct uct_tl_resource_desc`
  
  *Communication resource descriptor.*

- `struct uct_iface_attr`
  
  *Interface attributes: capabilities and limitations.*

  - `struct uct_iface_attr.cap`
  
    *List of capabilities supported by UCT iface event API.*

  - `struct uct_iface_attr.cap.put`
  
  - `struct uct_iface_attr.cap.get`
  
  - `struct uct_iface_attr.cap.am`
  
  - `struct uct_iface_attr.cap.tag`
  
  - `struct uct_iface_attr.cap.tag.recv`
  
  - `struct uct_iface_attr.cap.tag.eager`
  
  - `struct uct_iface_attr.cap.tag.rndv`
  
  - `struct uct_iface_attr.cap.atomic32`
  
  - `struct uct_iface_attr.cap.atomic64`

- `struct uct_iface_params`
  
  *Parameters used for interface creation.*

  - `union uct_iface_params.mode`
  
  - `struct uct_iface_params.mode.device`
  
  - `struct uct_iface_params.mode.sockaddr`

- `struct uct_ep_params`
  
  *Parameters for creating a UCT endpoint by uct_ep_create.*

  - `struct uct_completion`
  
    *Completion handle.*

  - `struct uct_pending_req`
  
    *Pending request.*

  - `struct uct_iov`
  
    *Structure for scatter-gather I/O.*

Typedefs

- `typedef struct uct_md_resource_desc uct_md_resource_desc_t`
  
  *Memory domain resource descriptor.*

- `typedef struct uct_component_attr uct_component_attr_t`
  
  *UCT component attributes.*

- `typedef struct uct_tl_resource_desc uct_tl_resource_desc_t`
  
  *Communication resource descriptor.*
typedef struct uct_component * uct_component_h
typedef struct uct_iface * uct_iface_h
typedef struct uct_iface_conf uct_iface_config_t
typedef struct uct_md_conf uct_md_config_t
typedef struct uct_cm_conf uct_cm_config_t
typedef struct uct_ep * uct_ep_h
typedef * uct_mem_h
typedef uintptr_t uct_rkey_t
typedef struct uct_md * uct_md_h
typedef struct uct_md_ops uct_md_ops_t
typedef struct uct_iface_attr uct_iface_attr_t
typedef struct uct_iface_params uct_iface_params_t
typedef struct uct_md_attr uct_md_attr_t
typedef struct uct_completion uct_completion_t
typedef struct uct_pending_req uct_pending_req_t
typedef struct uct_worker * uct_worker_h
typedef struct uct_md uct_md_t
typedef enum uct_am_trace_type uct_am_trace_type_t
typedef struct uct_device_addr uct_device_addr_t
typedef struct uct_iface_addr uct_iface_addr_t
typedef struct uct_ep_addr uct_ep_addr_t
typedef struct uct_ep_params uct_ep_params_t
typedef struct uct_cm_attr uct_cm_attr_t
typedef struct uct_cm uct_cm_t
typedef uct_cm_t * uct_cm_h
typedef struct uct_listener_attr uct_listener_attr_t
typedef struct uct_listener_params uct_listener_params_t
typedef struct uct_tag_context uct_tag_context_t
typedef uint64_t uct_tag_t
typedef uct_worker_cb_id_t
typedef void * uct_conn_request_h
typedef struct uct_iov uct_iov_t

copyright 2021 Unified Communication X (UCX). All rights reserved.
Enumerations

- enum uct_component_attr_field { UCT_COMPONENT_ATTR_FIELD_NAME = UCS_BIT(0), UCT_COMPONENT_ATTR_FIELD_MD_RESOURCES = UCS_BIT(2), UCT_COMPONENT_ATTR_FIELD_FLAGS = UCS_BIT(3) }  
  UCT component attributes field mask.

- enum { UCT_COMPONENT_FLAG_CM = UCS_BIT(0) }  
  Capability flags of uct_component_h.

- enum uct_device_type_t { UCT_DEVICE_TYPE_NET, UCT_DEVICE_TYPE_SHM, UCT_DEVICE_TYPE_ACC, UCT_DEVICE_TYPE_SELF, UCT_DEVICE_TYPE_LAST }  
  List of UCX device types.

- enum uct_iface_event_types { UCT_EVENT_SEND_COMP = UCS_BIT(0), UCT_EVENT_RECV = UCS_BIT(1), UCT_EVENT_RECV_SIG = UCS_BIT(2) }  
  Asynchronous event types.

- enum uct_flush_flags { UCT_FLUSH_FLAG_LOCAL = 0, UCT_FLUSH_FLAG_CANCEL = UCS_BIT(0) }  
  Flush modifiers.

- enum uct_progress_types { UCT_PROGRESS_SEND = UCS_BIT(0), UCT_PROGRESS_RECV = UCS_BIT(1), UCT_PROGRESS_THREAD_SAFE = UCS_BIT(7) }  
  UCT progress types.

- enum uct_cb_flags { UCT_CB_FLAG_RESERVED = UCS_BIT(1), UCT_CB_FLAG_ASYNC = UCS_BIT(2) }  
  Callback flags.

- enum uct_iface_open_mode { UCT_IFACE_OPEN_MODE_DEVICE = UCS_BIT(0), UCT_IFACE_OPEN_MODE_SOCKADDR_SERVER = UCS_BIT(1), UCT_IFACE_OPEN_MODE_SOCKADDR_CLIENT = UCS_BIT(2) }  
  Mode in which to open the interface.

- enum uct_iface_params_field { UCT_IFACE_PARAM_FIELD_CPU_MASK = UCS_BIT(0), UCT_IFACE_PARAM_FIELD_OPEN_MODE = UCS_BIT(1), UCT_IFACE_PARAM_FIELD_DEVICE = UCS_BIT(2), UCT_IFACE_PARAM_FIELD_SOCKADDR = UCS_BIT(3), UCT_IFACE_PARAM_FIELD_STATS_ROOT = UCS_BIT(4), UCT_IFACE_PARAM_FIELD_RX_HEADROOM = UCS_BIT(5), UCT_IFACE_PARAM_FIELD_ERR_HANDLER_ARG = UCS_BIT(6), UCT_IFACE_PARAM_FIELD_ERR_HANDLER = UCS_BIT(7), UCT_IFACE_PARAM_FIELD_ERR_HANDLER_FLAGS = UCS_BIT(8), UCT_IFACE_PARAM_FIELD_HW_TM_EAGER_ARG = UCS_BIT(9), UCT_IFACE_PARAM_FIELD_HW_TM_EAGER_CB = UCS_BIT(10), UCT_IFACE_PARAM_FIELD_HW_TM_RNDV_ARG = UCS_BIT(11), UCT_IFACE_PARAM_FIELD_HW_TM_RNDV_CB = UCS_BIT(12), UCT_IFACE_PARAM_FIELD_ASYNC_EVENT_ARG = UCS_BIT(13), UCT_IFACE_PARAM_FIELD_ASYNC_EVENT_CB = UCS_BIT(14), UCT_IFACE_PARAM_FIELD_KEEPALIVE_INTERVAL = UCS_BIT(15) }  
  UCT interface created by uct_iface_open parameters field mask.

  UCT endpoint created by uct_ep_create parameters field mask.

- enum uct_tag_context { UCT_TAG_RECV_CB_INLINE_DATA = UCS_BIT(0) }  
  Flags of uct_tag_context.

- enum uct_cb_param_flags { UCT_CB_PARAM_FLAG_DESC = UCS_BIT(0), UCT_CB_PARAM_FLAG_FIRST = UCS_BIT(1), UCT_CB_PARAM_FLAG_MORE = UCS_BIT(2) }  
  Flags for active message and tag-matching offload callbacks (callback's parameters).
Functions

- `ucs_status_t uct_query_components (uct_component_h **components_p, unsigned *num_components_p)`
  Query for list of components.
- `void uct_release_component_list (uct_component_h *components)`
  Release the list of components returned from `uct_query_components`.
- `ucs_status_t uct_component_query (uct_component_h component, uct_component_attr_t *component_attr)`
  Get component attributes.
- `ucs_status_t uct_md_open (uct_component_h component, const char *md_name, const uct_md_config_t *config, uct_md_h *md_p)`
  Open a memory domain.
- `void uct_md_close (uct_md_h md)`
  Close a memory domain.
- `ucs_status_t uct_md_query_tl_resources (uct_md_h md, uct_tl_resource_desc_t **resources_p, unsigned *num_resources_p)`
  Query for transport resources.
- `void uct_release_tl_resource_list (uct_tl_resource_desc_t *resources)`
  Release the list of resources returned from `uct_md_query_tl_resources`.
- `ucs_status_t uct_md_iface_config_read (uct_md_h md, const char *tl_name, const char *env_prefix, const char *filename, uct_iface_config_t **config_p)`
  Read transport-specific interface configuration.
- `void uct_config_release (void *config)`
  Release configuration memory returned from `uct_md_iface_config_read()`, `uct_md_config_read()`, or from `uct_cm_config_read()`.
- `ucs_status_t uct_iface_open (uct_md_h md, uct_worker_h worker, const uct_iface_params_t *params, const uct_iface_config_t *config, uct_iface_h *iface_p)`
  Open a communication interface.
- `void uct_iface_close (uct_iface_h iface)`
  Close and destroy an interface.
- `ucs_status_t uct_iface_query (uct_iface_h iface, uct_iface_attr_t *iface_attr)`
  Get interface attributes.
- `ucs_status_t uct_iface_get_device_address (uct_iface_h iface, uct_device_addr_t *addr)`
  Get address of the device the interface is using.
- `ucs_status_t uct_iface_get_address (uct_iface_h iface, uct_iface_addr_t *addr)`
  Get interface address.
- `int uct_iface_is_reachable (const uct_iface_h iface, const uct_device_addr_t *dev_addr, const uct_iface_addr_t *iface_addr)`
  Check if remote iface address is reachable.
- `ucs_status_t uct_ep_check (const uct_ep_h ep, unsigned flags, uct_completion_t *comp)`
  Check if the destination endpoint is alive in respect to UCT library.
- `ucs_status_t uct_iface_event_fd_get (uct_iface_h iface, int *fd_p)`
  Obtain a notification file descriptor for polling.
- `ucs_status_t uct_iface_event_arm (uct_iface_h iface, unsigned events)`
  Turn on event notification for the next event.
- `ucs_status_t uct_iface_mem_alloc (uct_iface_h iface, size_t length, unsigned flags, const char *name, uct_allocated_memory_t *mem)`
  Allocate memory which can be used for zero-copy communications.
- `void uct_iface_mem_free (const uct_allocated_memory_t *mem)`
  Release memory allocated with `uct_iface_mem_alloc()`.
- `ucs_status_t uct_ep_create (const uct_ep_params_t *params, uct_ep_h *ep_p)`
  Create new endpoint.
- `void uct_ep_destroy (uct_ep_h ep)`
  Release configuration memory returned from `uct_md_iface_config_read()`.
Destroy an endpoint.

- `ucs_status_t uct_ep_get_address (uct_ep_h ep, uct_ep_addr_t *addr)`
  Get endpoint address.

- `ucs_status_t uct_ep_connect_to_ep (uct_ep_h ep, const uct_device_addr_t *dev_addr, const uct_ep_addr_t *ep_addr)`
  Connect endpoint to a remote endpoint.

- `ucs_status_t uct_iface_flush (uct_iface_h iface, unsigned flags, uct_completion_t *comp)`
  Flush outstanding communication operations on an interface.

- `ucs_status_t uct_iface_fence (uct_iface_h iface, unsigned flags)`
  Ensures ordering of outstanding communications on the interface. Operations issued on the interface prior to this call are guaranteed to be completed before any subsequent communication operations to the same interface which follow the call to fence.

- `ucs_status_t uct_ep_pending_add (uct_ep_h ep, uct_pending_req_t *req, unsigned flags)`
  Add a pending request to an endpoint.

- `void uct_ep_pending_purge (uct_ep_h ep, uct_pending_purge_callback_t cb, void *arg)`
  Remove all pending requests from an endpoint.

- `ucs_status_t uct_ep_flush (uct_ep_h ep, unsigned flags, uct_completion_t *comp)`
  Flush outstanding communication operations on an endpoint.

- `ucs_status_t uct_iface_purge (uct_iface_h iface, unsigned flags)`
  Ensures ordering of outstanding communications on the endpoint. Operations issued on the endpoint prior to this call are guaranteed to be completed before any subsequent communication operations to the same endpoint which follow the call to fence.

- `void uct_iface_progress_enable (uct_iface_h iface, unsigned flags)`
  Enable synchronous progress for the interface.

- `void uct_iface_progress_disable (uct_iface_h iface, unsigned flags)`
  Disable synchronous progress for the interface.

- `unsigned uct_iface_progress (uct_iface_h iface)`
  Perform a progress on an interface.

- `static UCS_F_ALWAYS_INLINE void uct_completion_update_status (uct_completion_t *comp, ucs_status_t status)`
  Update status of UCT completion handle.

### 6.11.1 Detailed Description

This section describes a concept of the Communication Resource and routines associated with the concept.

### 6.11.2 Data Structure Documentation

#### 6.11.2.1 struct uct_md_resource_desc

This structure describes a memory domain resource.

**Data Fields**

<table>
<thead>
<tr>
<th>char</th>
<th>md_name[UCT_MD_NAME_MAX]</th>
<th>Memory domain name</th>
</tr>
</thead>
</table>

#### 6.11.2.2 struct uct_component_attr

This structure defines the attributes for UCT component. It is used for `uct_component_query`
### 6.11 UCT Communication Resource

Examples

```c
uct_hello_world.c.
```

#### Data Fields

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint64_t</td>
<td>field_mask</td>
<td>Mask of valid fields in this structure, using bits from <code>uct_component_attr_field</code>. Fields not specified in this mask will be ignored. Provides ABI compatibility with respect to adding new fields.</td>
</tr>
<tr>
<td>char</td>
<td>name[UCT_COMPONENT_NAME_MAX]</td>
<td>Component name</td>
</tr>
<tr>
<td>unsigned</td>
<td>md_resource_count</td>
<td>Number of memory-domain resources</td>
</tr>
<tr>
<td>uct_md_resource_desc_t*</td>
<td>md_resources</td>
<td>Array of memory domain resources. When used, it should be initialized prior to calling <code>uct_component_query</code> with a pointer to an array, which is large enough to hold all memory domain resource entries. After the call, this array will be filled with information about existing memory domain resources. In order to allocate this array, you can call <code>uct_component_query</code> twice: The first time would only obtain the amount of entries required, by specifying <code>UCT_COMPONENT_ATTR_FIELD_MD_RESOURCE_COUNT</code> in field_mask. Then the array could be allocated with the returned number of entries, and passed to a second call to <code>uct_component_query</code>, this time setting field_mask to <code>UCT_COMPONENT_ATTR_FIELD_MD_RESOURCES</code>.</td>
</tr>
<tr>
<td>uint64_t</td>
<td>flags</td>
<td>Flags as defined by <code>UCT_COMPONENT_FLAG_xx</code>.</td>
</tr>
</tbody>
</table>

#### 6.11.2.3 struct uct_tl_resource_desc

Resource descriptor is an object representing the network resource. Resource descriptor could represent a stand-alone communication resource such as an HCA port, network interface, or multiple resources such as multiple network interfaces or communication ports. It could also represent virtual communication resources that are defined over a single physical network interface.

Examples

```c
uct_hello_world.c.
```

#### Data Fields

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>tl_name[UCT_TL_NAME_MAX]</td>
<td>Transport name</td>
</tr>
<tr>
<td>char</td>
<td>dev_name[UCT_DEVICE_NAME_MAX]</td>
<td>Hardware device name</td>
</tr>
<tr>
<td>uct_device_type_t</td>
<td>dev_type</td>
<td>The device represented by this resource (e.g. UCT_DEVICE_TYPE_NET for a network interface)</td>
</tr>
</tbody>
</table>
Data Fields

| ucs_sys_device_t | sys_device | The identifier associated with the device bus_id as captured in ucs_sys_bus_id_t struct |

**6.11.2.4 struct uct_iface_attr**

Examples

uct_hello_world.c.

Data Fields

<table>
<thead>
<tr>
<th>struct uct_iface_attr</th>
<th>cap</th>
<th>Interface capabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>size_t device_addr_len</td>
<td>Size of device address</td>
<td></td>
</tr>
<tr>
<td>size_t iface_addr_len</td>
<td>Size of interface address</td>
<td></td>
</tr>
<tr>
<td>size_t ep_addr_len</td>
<td>Size of endpoint address</td>
<td></td>
</tr>
<tr>
<td>size_t max_conn_priv</td>
<td>Max size of the iface's private data. used for connection establishment with sockaddr</td>
<td></td>
</tr>
<tr>
<td>struct sockaddr_storage listen_sockaddr</td>
<td>Sockaddr on which this iface is listening.</td>
<td></td>
</tr>
<tr>
<td>double overhead</td>
<td>Message overhead, seconds</td>
<td></td>
</tr>
<tr>
<td>uct_ppn_bandwidth_t</td>
<td>bandwidth</td>
<td>Bandwidth model</td>
</tr>
<tr>
<td>ucs_linear_func_t</td>
<td>latency</td>
<td>Latency as function of number of active endpoints</td>
</tr>
<tr>
<td>uint8_t priority</td>
<td>Priority of device</td>
<td></td>
</tr>
<tr>
<td>size_t max_num_eps</td>
<td>Maximum number of endpoints</td>
<td></td>
</tr>
<tr>
<td>unsigned dev_num_paths</td>
<td>How many network paths can be utilized on the device used by this interface for optimal performance. Endpoints that connect to the same remote address but use different paths can potentially achieve higher total bandwidth compared to using only a single endpoint.</td>
<td></td>
</tr>
</tbody>
</table>

**6.11.2.5 struct uct_iface_attr.cap**

Data Fields

<table>
<thead>
<tr>
<th>cap</th>
<th>put</th>
<th>Attributes for PUT operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>cap</td>
<td>get</td>
<td>Attributes for GET operations</td>
</tr>
<tr>
<td>cap</td>
<td>am</td>
<td>Attributes for AM operations</td>
</tr>
<tr>
<td>cap</td>
<td>tag</td>
<td>Attributes for TAG operations</td>
</tr>
<tr>
<td>cap</td>
<td>atomic32</td>
<td></td>
</tr>
<tr>
<td>cap</td>
<td>atomic64</td>
<td>Attributes for atomic operations</td>
</tr>
<tr>
<td>uint64_t flags</td>
<td>Flags from UCT interface operations and capabilities</td>
<td></td>
</tr>
<tr>
<td>uint64_t event_flags</td>
<td>Flags from UCT interface for asynchronous event capabilities</td>
<td></td>
</tr>
</tbody>
</table>

**6.11.2.6 struct uct_iface_attr.cap.put**

Data Fields

| size_t | max_short | Maximal size for put_short |

© 2021 Unified Communication X (UCX). All rights reserved.
### Data Fields

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>max_bcopy</code></td>
<td>Maximal size for <code>put_bcopy</code></td>
</tr>
<tr>
<td><code>min_zcopy</code></td>
<td>Minimal size for <code>put_zcopy</code> (total of <code>uct_iov_t::length</code> of the <code>iov</code> parameter)</td>
</tr>
<tr>
<td><code>max_zcopy</code></td>
<td>Maximal size for <code>put_zcopy</code> (total of <code>uct_iov_t::length</code> of the <code>iov</code> parameter)</td>
</tr>
<tr>
<td><code>opt_zcopy_align</code></td>
<td>Optimal alignment for zero-copy buffer address</td>
</tr>
<tr>
<td><code>align_mtu</code></td>
<td>MTU used for alignment</td>
</tr>
<tr>
<td><code>max_iov</code></td>
<td>Maximal <code>iovcnt</code> parameter in <code>uct_ep_put_zcopy</code></td>
</tr>
</tbody>
</table>

### 6.11.2.7 struct uct_iface_attr.cap.get

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>max_short</code></td>
<td>Maximal size for <code>get_short</code></td>
</tr>
<tr>
<td><code>max_bcopy</code></td>
<td>Maximal size for <code>get_bcopy</code></td>
</tr>
<tr>
<td><code>min_zcopy</code></td>
<td>Minimal size for <code>get_zcopy</code> (total of <code>uct_iov_t::length</code> of the <code>iov</code> parameter)</td>
</tr>
<tr>
<td><code>max_zcopy</code></td>
<td>Maximal size for <code>get_zcopy</code> (total of <code>uct_iov_t::length</code> of the <code>iov</code> parameter)</td>
</tr>
<tr>
<td><code>opt_zcopy_align</code></td>
<td>Optimal alignment for zero-copy buffer address</td>
</tr>
<tr>
<td><code>align_mtu</code></td>
<td>MTU used for alignment</td>
</tr>
<tr>
<td><code>max_iov</code></td>
<td>Maximal <code>iovcnt</code> parameter in <code>uct_ep_get_zcopy</code></td>
</tr>
</tbody>
</table>

### 6.11.2.8 struct uct_iface_attr.cap.am

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>max_short</code></td>
<td>Total max. size (incl. the header)</td>
</tr>
<tr>
<td><code>max_bcopy</code></td>
<td>Total max. size (incl. the header)</td>
</tr>
<tr>
<td><code>min_zcopy</code></td>
<td>Minimal size for am_zcopy (incl. the header and total of <code>uct_iov_t::length</code> of the <code>iov</code> parameter)</td>
</tr>
<tr>
<td><code>max_zcopy</code></td>
<td>Total max. size (incl. the header and total of <code>uct_iov_t::length</code> of the <code>iov</code> parameter)</td>
</tr>
<tr>
<td><code>opt_zcopy_align</code></td>
<td>Optimal alignment for zero-copy buffer address</td>
</tr>
<tr>
<td><code>align_mtu</code></td>
<td>MTU used for alignment</td>
</tr>
<tr>
<td><code>max_hdr</code></td>
<td>Max. header size for zcopy</td>
</tr>
<tr>
<td><code>max_iov</code></td>
<td>Maximal <code>iovcnt</code> parameter in <code>uct_ep_am_zcopy</code></td>
</tr>
</tbody>
</table>

### 6.11.2.9 struct uct_iface_attr.cap.tag

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>recv</code></td>
<td>Attributes related to eager protocol</td>
</tr>
<tr>
<td><code>eager</code></td>
<td>Attributes related to rendezvous protocol</td>
</tr>
</tbody>
</table>

### 6.11.2.10 struct uct_iface_attr.tag.recv

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>min_recv</code></td>
<td>Minimal allowed length of posted receive buffer</td>
</tr>
<tr>
<td><code>max_zcopy</code></td>
<td>Maximal allowed data length in <code>uct_iface_tag_recv_zcopy</code></td>
</tr>
</tbody>
</table>
### Data Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>size_t</code> max_iov</td>
<td>Maximal <code>iovcnt</code> parameter in <code>uct_iface_tag_recv_zcopy</code></td>
</tr>
<tr>
<td><code>size_t</code> max_outstanding</td>
<td>Maximal number of simultaneous receive operations</td>
</tr>
</tbody>
</table>

#### 6.11.2.11 struct uct_iface_attr.cap.tag.eager

### Data Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>size_t</code> max_short</td>
<td>Maximal allowed data length in <code>uct_ep_tag_eager_short</code></td>
</tr>
<tr>
<td><code>size_t</code> max_bcopy</td>
<td>Maximal allowed data length in <code>uct_ep_tag_eager_bcopy</code></td>
</tr>
<tr>
<td><code>size_t</code> max_zcopy</td>
<td>Maximal allowed data length in <code>uct_ep_tag_eager_zcopy</code></td>
</tr>
<tr>
<td><code>size_t</code> max_iov</td>
<td>Maximal <code>iovcnt</code> parameter in <code>uct_ep_tag_eager_zcopy</code></td>
</tr>
</tbody>
</table>

#### 6.11.2.12 struct uct_iface_attr.cap.tag.rndv

### Data Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>size_t</code> max_zcopy</td>
<td>Maximal allowed data length in <code>uct_ep_tag_rndv_zcopy</code></td>
</tr>
<tr>
<td><code>size_t</code> max_hdr</td>
<td>Maximal allowed header length in <code>uct_ep_tag_rndv_zcopy</code> and <code>uct_ep_tag_rndv_request</code></td>
</tr>
<tr>
<td><code>size_t</code> max_iov</td>
<td>Maximal <code>iovcnt</code> parameter in <code>uct_ep_tag_rndv_zcopy</code></td>
</tr>
</tbody>
</table>

#### 6.11.2.13 struct uct_iface_attr.cap.atomic32

### Data Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uint64_t</code> op_flags</td>
<td>Attributes for atomic-post operations</td>
</tr>
<tr>
<td><code>uint64_t</code> fop_flags</td>
<td>Attributes for atomic-fetch operations</td>
</tr>
</tbody>
</table>

#### 6.11.2.14 struct uct_iface_attr.cap.atomic64

### Data Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uint64_t</code> op_flags</td>
<td>Attributes for atomic-post operations</td>
</tr>
<tr>
<td><code>uint64_t</code> fop_flags</td>
<td>Attributes for atomic-fetch operations</td>
</tr>
</tbody>
</table>

#### 6.11.2.15 struct uct_iface_params

This structure should be allocated by the user and should be passed to `uct_iface_open`. User has to initialize all fields of this structure.

**Examples**

- `uct_hello_world.c`

### Data Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uint64_t</code> field_mask</td>
<td>Mask of valid fields in this structure, using bits from <code>uct_iface_params_field</code>. Fields not specified in this mask will be ignored.</td>
</tr>
</tbody>
</table>
### 6.11 UCT Communication Resource

#### Data Fields

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ucs_cpu_set_t</code></td>
<td>Mask of CPUs to use for resources</td>
</tr>
<tr>
<td><code>uint64_t</code></td>
<td>Interface open mode bitmap.</td>
</tr>
<tr>
<td><code>union uct_iface_params</code></td>
<td>Mode-specific parameters</td>
</tr>
<tr>
<td><code>ucs_stats_node_t *</code></td>
<td>Root in the statistics tree. Can be NULL. If non NULL, it will be a root of <code>uct_iface</code> object in the statistics tree.</td>
</tr>
<tr>
<td><code>size_t</code></td>
<td>How much bytes to reserve before the receive segment.</td>
</tr>
<tr>
<td><code>void *</code></td>
<td>Custom argument of <code>err_handler</code>.</td>
</tr>
<tr>
<td><code>uct_error_handler_t</code></td>
<td>The callback to handle transport level error.</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>Callback flags to indicate where the <code>err_handler</code> callback can be invoked from.</td>
</tr>
<tr>
<td><code>void *</code></td>
<td>These callbacks are only relevant for HW Tag Matching.</td>
</tr>
<tr>
<td><code>uct_tag_unexp_eager_cb_t</code></td>
<td>Callback for tag matching unexpected eager messages.</td>
</tr>
<tr>
<td><code>void *</code></td>
<td>Callback for tag matching unexpected rndv messages.</td>
</tr>
<tr>
<td><code>uct_async_event_cb_t</code></td>
<td>Callback for asynchronous event handling.</td>
</tr>
<tr>
<td><code>ucs_time_t</code></td>
<td>Keepalive interval</td>
</tr>
</tbody>
</table>

#### 6.11.2.16 union uct_iface_params.mode

Mode-specific parameters

#### Data Fields

<table>
<thead>
<tr>
<th>Mode</th>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mode</code></td>
<td><code>device</code></td>
</tr>
<tr>
<td></td>
<td>The fields in this structure (tl_name and dev_name) need to be set only when the <code>UCT_IFACE_OPEN_MODE_DEVICE</code> bit is set in <code>uct_iface_params_t::open_mode</code>. This will make <code>uct_iface_open</code> open the interface on the specified device.</td>
</tr>
<tr>
<td><code>mode</code></td>
<td><code>sockaddr</code></td>
</tr>
<tr>
<td></td>
<td>These callbacks and address are only relevant for client-server connection establishment with sockaddr and are needed on the server side. The callbacks and address need to be set when the <code>UCT_IFACE_OPEN_MODE_SOCKADDR_SERVER</code> bit is set in <code>uct_iface_params_t::open_mode</code>. This will make <code>uct_iface_open</code> open the interface on the specified address as a server.</td>
</tr>
</tbody>
</table>

#### 6.11.2.17 struct uct_iface_params.mode.device

The fields in this structure (tl_name and dev_name) need to be set only when the `UCT_IFACE_OPEN_MODE_DEVICE` bit is set in `uct_iface_params_t::open_mode`. This will make `uct_iface_open` open the interface on the specified device.

#### Data Fields

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>const char *</code></td>
<td><code>tl_name</code></td>
</tr>
<tr>
<td><code>const char *</code></td>
<td><code>dev_name</code></td>
</tr>
</tbody>
</table>
6.11.2.18 struct uct_iface_params.mode.sockaddr

These callbacks and address are only relevant for client-server connection establishment with sockaddr and are needed on the server side. The callbacks and address need to be set when the UCT_IFACE_OPEN_MODE_SOCKETADDR_SERVER bit is set in uct_iface_params_t::open_mode. This will make uct_iface_open open the interface on the specified address as a server.

Data Fields

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ucs_sock_addr_t</td>
<td>listen_sockaddr</td>
<td></td>
</tr>
<tr>
<td>void ∗</td>
<td>conn_request_arg</td>
<td>Argument for connection request callback</td>
</tr>
<tr>
<td>uct_sockaddr_conn_request_callback ∗</td>
<td>conn_request_cb</td>
<td>Callback for an incoming connection request on the server</td>
</tr>
<tr>
<td>uint32_t</td>
<td>cb_flags</td>
<td>Callback flags to indicate where the callback can be invoked from. uct_cb_flags</td>
</tr>
</tbody>
</table>

6.11.2.19 struct uct_ep_params

Examples

uct_hello_world.c.

Data Fields

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint64_t</td>
<td>field_mask</td>
<td>Mask of valid fields in this structure, using bits from uct_ep_params_field. Fields not specified by this mask will be ignored.</td>
</tr>
<tr>
<td>uct_iface_t</td>
<td>iface</td>
<td>Interface to create the endpoint on. Either iface or cm field must be initialized but not both.</td>
</tr>
<tr>
<td>void ∗</td>
<td>user_data</td>
<td>User data associated with the endpoint.</td>
</tr>
<tr>
<td>const uct_device_addr_t ∗</td>
<td>dev_addr</td>
<td>The device address to connect to on the remote peer. This must be defined together with uct_ep_params_t::iface_addr to create an endpoint connected to a remote interface.</td>
</tr>
<tr>
<td>const uct_iface_addr_t ∗</td>
<td>iface_addr</td>
<td>This specifies the remote address to use when creating an endpoint that is connected to a remote interface. Note This requires UCT_IFACE_FLAG_CONNECT_TO_IFACE capability.</td>
</tr>
<tr>
<td>const ucs_sock_addr_t ∗</td>
<td>sockaddr</td>
<td>The sockaddr to connect to on the remote peer. If set, uct_ep_create will create an endpoint for a connection to the remote peer, specified by its socket address. Note The interface in this routine requires the UCT_IFACE_FLAG_CONNECT_TO_SOCKADDR capability.</td>
</tr>
</tbody>
</table>
### Data Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uint32_t sockaddr_cb_flags</code></td>
<td><code>uct_cb_flags</code> to indicate <code>uct_ep_params_t::sockaddr_pack_cb</code> behavior. If <code>uct_ep_params_t::sockaddr_pack_cb</code> is not set, this field will be ignored.</td>
</tr>
<tr>
<td><code>uct_cm_ep_priv_data_pack_callback_t</code></td>
<td>Callback that will be used for filling the user's private data to be delivered to the remote peer by the callback on the server or client side. This field is only valid if <code>uct_ep_params_t::sockaddr</code> is set. Note It is never guaranteed that the callback will be called. If, for example, the endpoint goes into error state before issuing the connection request, the callback will not be invoked.</td>
</tr>
<tr>
<td><code>uct_cm_h</code></td>
<td>The connection manager object as created by <code>uct_cm_open</code>. Either <code>cm</code> or <code>iface</code> field must be initialized but not both.</td>
</tr>
<tr>
<td><code>uct_conn_request_h</code></td>
<td>Connection request that was passed to <code>uct_cm_listener_conn_request_args_t::conn_request</code>. Note After a call to <code>uct_ep_create</code>, <code>params.conn_request</code> is consumed and should not be used anymore, even if the call returns with an error.</td>
</tr>
<tr>
<td><code>uct_cm_ep_client_connect_callback_t</code></td>
<td>Callback that will be invoked when the endpoint on the client side is being connected to the server by a connection manager <code>uct_cm_h</code>.</td>
</tr>
<tr>
<td><code>uct_cm_ep_server_conn_notify_callback_t</code></td>
<td>Callback that will be invoked when the endpoint on the server side is being connected to a client by a connection manager <code>uct_cm_h</code>.</td>
</tr>
<tr>
<td><code>uct_ep_disconnect_cb_t</code></td>
<td>Callback that will be invoked when the endpoint is disconnected.</td>
</tr>
<tr>
<td><code>unsigned path_index</code></td>
<td>Index of the path which the endpoint should use, must be in the range 0...(uct_iface_attr_t::dev_num_paths - 1).</td>
</tr>
</tbody>
</table>

### 6.11.2.20 struct uct_completion

This structure should be allocated by the user and can be passed to communication primitives. The user must initialize all fields of the structure. If the operation returns UCS_INPROGRESS, this structure will be in use by the transport until the operation completes. When the operation completes, "count" field is decremented by 1, and whenever it reaches 0 - the callback is called.

Notes:
• The same structure can be passed multiple times to communication functions without the need to wait for completion.

• If the number of operations is smaller than the initial value of the counter, the callback will not be called at all, so it may be left undefined.

• status field is required to track the first time the error occurred, and report it via a callback when count reaches 0.

Examples

uct_hello_world.c.

Data Fields

<table>
<thead>
<tr>
<th>uct_completion_callback_t</th>
<th>func</th>
<th>User callback function</th>
</tr>
</thead>
<tbody>
<tr>
<td>int count</td>
<td></td>
<td>Completion counter</td>
</tr>
<tr>
<td>ucs_status_t status</td>
<td></td>
<td>Completion status, this field must be initialized with UCS_OK before first operation is started.</td>
</tr>
</tbody>
</table>

6.11.2.21 struct uct_pending_req

This structure should be passed to uct_ep_pending_add() and is used to signal new available resources back to user.

Data Fields

<table>
<thead>
<tr>
<th>uct_pending_callback_t</th>
<th>func</th>
<th>User callback function</th>
</tr>
</thead>
<tbody>
<tr>
<td>char priv[UCT_PENDING_REQ_PRIV_LEN]</td>
<td></td>
<td>Used internally by UCT</td>
</tr>
</tbody>
</table>

6.11.2.22 struct uct_iov

Specifies a list of buffers which can be used within a single data transfer function call.

<table>
<thead>
<tr>
<th>buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>+-----+-----+-----+-----+-----+-----+-----+-----</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>+-----+-----+-----+-----+-----+-----+-----+-----</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Note

The sum of lengths in all iov list must be less or equal to max_zcopy of the respective communication operation.
If length or count are zero, the memory pointed to by buffer will not be accessed. Otherwise, buffer must point to valid memory.
If count is one, every iov entry specifies a single contiguous data block
If count > 1, each iov entry specifies a strided block of count elements and distance of stride byte between consecutive elements

Examples

uct_hello_world.c.
Data Fields

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void *</code></td>
<td>buffer</td>
<td>Data buffer</td>
</tr>
<tr>
<td><code>size_t</code></td>
<td>length</td>
<td>Length of the payload in bytes</td>
</tr>
<tr>
<td><code>uct_mem_h</code></td>
<td>memh</td>
<td>Local memory key descriptor for the data</td>
</tr>
<tr>
<td><code>size_t</code></td>
<td>stride</td>
<td>Stride between beginnings of payload elements in the buffer in bytes</td>
</tr>
<tr>
<td><code>unsigned</code></td>
<td>count</td>
<td>Number of payload elements in the buffer</td>
</tr>
</tbody>
</table>

6.11.3 Typedef Documentation

6.11.3.1 `uct_md_resource_desc_t`

```c
typedef struct uct_md_resource_desc uct_md_resource_desc_t
```

This structure describes a memory domain resource.

6.11.3.2 `uct_component_attr_t`

```c
typedef struct uct_component_attr uct_component_attr_t
```

This structure defines the attributes for UCT component. It is used for `uct_component_query`.

6.11.3.3 `uct_tl_resource_desc_t`

```c
typedef struct uct_tl_resource_desc uct_tl_resource_desc_t
```

Resource descriptor is an object representing the network resource. Resource descriptor could represent a stand-alone communication resource such as an HCA port, network interface, or multiple resources such as multiple network interfaces or communication ports. It could also represent virtual communication resources that are defined over a single physical network interface.

6.11.3.4 `uct_component_h`

```c
typedef struct uct_component* uct_component_h
```

6.11.3.5 `uct_iface_h`

```c
typedef struct uct_iface* uct_iface_h
```

6.11.3.6 `uct_iface_config_t`

```c
typedef struct uct_iface_config uct_iface_config_t
```

© 2021 Unified Communication X (UCX). All rights reserved.
6.11.3.7 uct_md_config_t

typedef struct uct_md_config uct_md_config_t

6.11.3.8 uct_cm_config_t

typedef struct uct_cm_config uct_cm_config_t

6.11.3.9 uct_ep_h

typedef struct uct_ep* uct_ep_h

6.11.3.10 uct_mem_h

typedef void* uct_mem_h

6.11.3.11 uct_rkey_t

typedef uintptr_t uct_rkey_t

6.11.3.12 uct_md_h

typedef struct uct_md* uct_md_h

6.11.3.13 uct_md_ops_t

typedef struct uct_md_ops uct_md_ops_t

6.11.3.14 uct_rkey_ctx_h

typedef void* uct_rkey_ctx_h

6.11.3.15 uct_iface_attr_t

typedef struct uct_iface_attr uct_iface_attr_t
6.11.3.16 uct_iface_params_t

typedef struct uct_iface_params uct_iface_params_t

6.11.3.17 uct_md_attr_t

typedef struct uct_md_attr uct_md_attr_t

6.11.3.18 uct_completion_t

typedef struct uct_completion uct_completion_t

6.11.3.19 uct_pending_req_t

typedef struct uct_pending_req uct_pending_req_t

6.11.3.20 uct_worker_h

typedef struct uct_worker* uct_worker_h

6.11.3.21 uct_md_t

typedef struct uct_md uct_md_t

6.11.3.22 uct_am_trace_type_t

typedef enum uct_am_trace_type uct_am_trace_type_t

6.11.3.23 uct_device_addr_t

typedef struct uct_device_addr uct_device_addr_t

6.11.3.24 uct_iface_addr_t

typedef struct uct_iface_addr uct_iface_addr_t
6.11.3.25 uct_ep_addr_t

typedef struct uct_ep_addr uct_ep_addr_t

6.11.3.26 uct_ep_params_t

typedef struct uct_ep_params uct_ep_params_t

6.11.3.27 uct_cm_attr_t

typedef struct uct_cm_attr uct_cm_attr_t

6.11.3.28 uct_cm_t

typedef struct uct_cm uct_cm_t

6.11.3.29 uct_cm_h

typedef uct_cm_t* uct_cm_h

6.11.3.30 uct_listener_attr_t

typedef struct uct_listener_attr uct_listener_attr_t

6.11.3.31 uct_listener_h

typedef struct uct_listener* uct_listener_h

6.11.3.32 uct_listener_params_t

typedef struct uct_listener_params uct_listener_params_t

6.11.3.33 uct_tag_context_t

typedef struct uct_tag_context uct_tag_context_t
6.11.3.34 **uct_tag_t**

typedef uint64_t uct_tag_t

6.11.3.35 **uct_worker_cb_id_t**

typedef int uct_worker_cb_id_t

6.11.3.36 **uct_conn_request_h**

typedef void* uct_conn_request_h

6.11.3.37 **uct_iov_t**

typedef struct uct_iov uct_iov_t

Specifies a list of buffers which can be used within a single data transfer function call.

```
+-----------+-------+-----------+-------+-----------+
| payload   | empty | payload   | empty | payload   |
|<length-->|       |<length-->|       |<length-->|
|<---- stride ----->|<---- stride ----->|
```

**Note**

The sum of lengths in all iov list must be less or equal to max_zcopy of the respective communication operation.

If `length` or `count` are zero, the memory pointed to by `buffer` will not be accessed. Otherwise, `buffer` must point to valid memory.

If `count` is one, every iov entry specifies a single contiguous data block.

If `count` > 1, each iov entry specifies a strided block of `count` elements and distance of `stride` byte between consecutive elements.

6.11.3.38 **uct_completion_callback_t**

typedef void(* uct_completion_callback_t) (uct_completion_t *self)

**Parameters**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>self</td>
<td>Pointer to relevant completion structure, which was initially passed to the operation.</td>
</tr>
</tbody>
</table>
6.11.3.39 uct_pending_callback_t

typedef ucs_status_t(* uct_pending_callback_t) (uct_pending_req_t *self)

Parameters

| in | self | Pointer to relevant pending structure, which was initially passed to the operation. |

Returns

- UCS_OK - This pending request has completed and should be removed.
- UCS_INPROGRESS - Some progress was made, but not completed. Keep this request and keep processing the queue. Otherwise - Could not make any progress. Keep this pending request on the queue, and stop processing the queue.

6.11.3.40 uct_error_handler_t

typedef ucs_status_t(* uct_error_handler_t) (void *arg, uct_ep_h ep, ucs_status_t status)

Parameters

| in | arg | User argument to be passed to the callback. |
| in | ep | Endpoint which has failed. Upon return from the callback, this ep is no longer usable and all subsequent operations on this ep will fail with the error code passed in status. |
| in | status | Status indicating error. |

Returns

- UCS_OK - The error was handled successfully. Otherwise - The error was not handled and is returned back to the transport.

6.11.3.41 uct_pending_purge_callback_t

typedef void(* uct_pending_purge_callback_t) (uct_pending_req_t *self, void *arg)

Parameters

| in | self | Pointer to relevant pending structure, which was initially passed to the operation. |
| in | arg | User argument to be passed to the callback. |

6.11.3.42 uct_pack_callback_t

typedef size_t(* uct_pack_callback_t) (void *dest, void *arg)

Parameters

| in | dest | Memory buffer to pack the data to. |
| in | arg | Custom user-argument. |
Returns

Size of the data was actually produced.

6.11.3.43 uct_unpack_callback_t

typedef void(∗uct_unpack_callback_t)(void ∗arg, const void ∗data, size_t length)

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>arg</th>
<th>Custom user-argument.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>data</td>
<td>Memory buffer to unpack the data from.</td>
</tr>
<tr>
<td>in</td>
<td>length</td>
<td>How much data to consume (size of &quot;data&quot;)</td>
</tr>
</tbody>
</table>

Note

The arguments for this callback are in the same order as libc's memcpy().

6.11.3.44 uct_async_event_cb_t

typedef void(∗uct_async_event_cb_t) (void ∗arg, unsigned flags)

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>arg</th>
<th>User argument to be passed to the callback.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>flags</td>
<td>Flags to be passed to the callback (reserved for future use).</td>
</tr>
</tbody>
</table>

6.11.4 Enumeration Type Documentation

6.11.4.1 uct_component_attr_field

enum uct_component_attr_field

The enumeration allows specifying which fields in uct_component_attr_t are present. It is used for backward compatibility support.

Enumerator

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCT_COMPONENT_ATTR_FIELD_NAME</td>
<td>Component name</td>
</tr>
<tr>
<td>UCT_COMPONENT_ATTR_FIELD_MD_RESOURCE_COUNT</td>
<td>MD resource count</td>
</tr>
<tr>
<td>UCT_COMPONENT_ATTR_FIELD_MD_RESOURCES</td>
<td>MD resources array</td>
</tr>
<tr>
<td>UCT_COMPONENT_ATTR_FIELD_FLAGS</td>
<td>Capability flags</td>
</tr>
</tbody>
</table>
6.11.4.2 anonymous enum

The enumeration defines bit mask of `uct_component_h` capabilities in `uct_component_attr_t::flags` which is set by `uct_component_query`.

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCT_COMPONENT_FLAG_CM</td>
<td>If set, the component supports <code>uct_cm_h</code> functionality. See <code>uct_cm_open</code> for details.</td>
</tr>
</tbody>
</table>

6.11.4.3 uct_device_type_t

```
enum uct_device_type_t
```

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCT_DEVICE_TYPE_NET</td>
<td>Network devices</td>
</tr>
<tr>
<td>UCT_DEVICE_TYPE_SHM</td>
<td>Shared memory devices</td>
</tr>
<tr>
<td>UCT_DEVICE_TYPE_ACC</td>
<td>Acceleration devices</td>
</tr>
<tr>
<td>UCT_DEVICE_TYPE_SELF</td>
<td>Loop-back device</td>
</tr>
<tr>
<td>UCT_DEVICE_TYPE_LAST</td>
<td></td>
</tr>
</tbody>
</table>

6.11.4.4 uct_iface_event_types

```
enum uct_iface_event_types
```

Note

The `UCT_EVENT_RECV` and `UCT_EVENT_RECV_SIG` event types are used to indicate receive-side completions for both tag matching and active messages. If the interface supports signaled receives (UCT_IFACE_FLAG_EVENT_RECV_SIG), then for the messages sent with UCT_SEND_FLAG_SIGNALED flag, `UCT_EVENT_RECV_SIG` should be triggered on the receiver. Otherwise, `UCT_EVENT_RECV` should be triggered.

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCT_EVENT_SEND_COMP</td>
<td>Send completion event</td>
</tr>
<tr>
<td>UCT_EVENT_RECV</td>
<td>Tag or active message received</td>
</tr>
<tr>
<td>UCT_EVENT_RECV_SIG</td>
<td>Signaled tag or active message received</td>
</tr>
</tbody>
</table>

6.11.4.5 uct_flush_flags

```
enum uct_flush_flags
```
### Enumerator

<table>
<thead>
<tr>
<th>UCT_FLUSH_FLAG_LOCAL</th>
<th>Guarantees that the data transfer is completed but the target buffer may not be updated yet.</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCT_FLUSH_FLAG_CANCEL</td>
<td>The library will make a best effort attempt to cancel all uncompleted operations. However, there is a chance that some operations will not be canceled in which case the user will need to handle their completions through the relevant callbacks. After <code>uct_ep_flush</code> with this flag is completed, the endpoint will be set to error state, and it becomes unusable for send operations and should be destroyed.</td>
</tr>
</tbody>
</table>

#### 6.11.4.6 `uct_progress_types`

**Enumerator**

<table>
<thead>
<tr>
<th>UCT_PROGRESS_SEND</th>
<th>Progress send operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCT_PROGRESS_RECV</td>
<td>Progress receive operations</td>
</tr>
<tr>
<td>UCT_PROGRESS_THREAD_SAFE</td>
<td>Enable/disable progress while another thread may be calling <code>ucp_worker_progress()</code>.</td>
</tr>
</tbody>
</table>

#### 6.11.4.7 `uct_cb_flags`

**Enumerator**

<table>
<thead>
<tr>
<th>UCT_CB_FLAG_RESERVED</th>
<th>Reserved for future use.</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCT_CB_FLAG_ASYNC</td>
<td>Callback is allowed to be called from any thread in the process, and therefore should be thread-safe. For example, it may be called from a transport async progress thread. To guarantee async invocation, the interface must have the <code>UCT_IFACE_FLAG_CB_ASYNC</code> flag set. If async callback is requested on an interface which only supports sync callback (i.e., only the <code>UCT_IFACE_FLAG_CB_SYNC</code> flag is set), the callback will be invoked only from the context that called <code>uct_iface_progress</code>).</td>
</tr>
</tbody>
</table>

#### 6.11.4.8 `uct_iface_open_mode`

**Enumerator**

| UCT_IFACE_OPEN_MODEDEVICE | Interface is opened on a specific device |
### 6.11.4.9 uct_iface_params_field

The enumeration allows specifying which fields in `uct_iface_params_t` are present, for backward compatibility support.

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCT_IFACE_PARAM_FIELD_CPU_MASK</td>
<td>Enables <code>uct_iface_params_t::cpu_mask</code></td>
</tr>
<tr>
<td>UCT_IFACE_PARAM_FIELD_OPEN_MODE</td>
<td>Enables <code>uct_iface_params_t::open_mode</code></td>
</tr>
<tr>
<td>UCT_IFACE_PARAM_FIELD_DEVICE</td>
<td>Enables <code>uct_iface_params_t::mode::device</code></td>
</tr>
<tr>
<td>UCT_IFACE_PARAM_FIELD_SOCKADDR</td>
<td>Enables <code>uct_iface_params_t::mode::sockaddr</code></td>
</tr>
<tr>
<td>UCT_IFACE_PARAM_FIELD_STATS_ROOT</td>
<td>Enables <code>uct_iface_params_t::stats_root</code></td>
</tr>
<tr>
<td>UCT_IFACE_PARAM_FIELD_RX_HEADROOM</td>
<td>Enables <code>uct_iface_params_t::rx_headroom</code></td>
</tr>
<tr>
<td>UCT_IFACE_PARAM_FIELD_ERR_HANDLER_ARG</td>
<td>Enables <code>uct_iface_params_t::err_handler_arg</code></td>
</tr>
<tr>
<td>UCT_IFACE_PARAM_FIELD_ERR_HANDLER</td>
<td>Enables <code>uct_iface_params_t::err_handler</code></td>
</tr>
<tr>
<td>UCT_IFACE_PARAM_FIELD_ERR_HANDLER_FLAGS</td>
<td>Enables <code>uct_iface_params_t::err_handler_flags</code></td>
</tr>
<tr>
<td>UCT_IFACE_PARAM_FIELD_HW_TM_EAGER_ARG</td>
<td>Enables <code>uct_iface_params_t::eager_arg</code></td>
</tr>
<tr>
<td>UCT_IFACE_PARAM_FIELD_HW_TM_EAGER_CB</td>
<td>Enables <code>uct_iface_params_t::eager_cb</code></td>
</tr>
<tr>
<td>UCT_IFACE_PARAM_FIELD_HW_TM_RNDV_ARG</td>
<td>Enables <code>uct_iface_params_t::rndv_arg</code></td>
</tr>
<tr>
<td>UCT_IFACE_PARAM_FIELD_HW_TM_RNDV_CB</td>
<td>Enables <code>uct_iface_params_t::rndv_cb</code></td>
</tr>
<tr>
<td>UCT_IFACE_PARAM_FIELD_ASYNC_EVENT_ARG</td>
<td>Enables <code>uct_iface_params_t::async_event_arg</code></td>
</tr>
<tr>
<td>UCT_IFACE_PARAM_FIELD_ASYNC_EVENT_CB</td>
<td>Enables <code>uct_iface_params_t::async_event_cb</code></td>
</tr>
<tr>
<td>UCT_IFACE_PARAM_FIELD_KEEPALIVE_INTERVAL</td>
<td>Enables <code>uct_iface_params_t::keepalive_interval</code></td>
</tr>
</tbody>
</table>

### 6.11.4.10 uct_ep_params_field

The enumeration allows specifying which fields in `uct_ep_params_t` are present, for backward compatibility support.

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCT_EP_PARAM_FIELD_IFACE</td>
<td>Enables <code>uct_ep_params::iface</code></td>
</tr>
<tr>
<td>UCT_EP_PARAM_FIELD_USER_DATA</td>
<td>Enables <code>uct_ep_params::user_data</code></td>
</tr>
<tr>
<td>UCT_EP_PARAM_FIELD_DEV_ADDR</td>
<td>Enables <code>uct_ep_params::dev_addr</code></td>
</tr>
<tr>
<td>UCT_EP_PARAM_FIELD_IFACE_ADDR</td>
<td>Enables <code>uct_ep_params::iface_addr</code></td>
</tr>
<tr>
<td>UCT_EP_PARAM_FIELD_SOCKADDR</td>
<td>Enables <code>uct_ep_params::sockaddr</code></td>
</tr>
</tbody>
</table>
6.11 UCT Communication Resource

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Enables uct_ep_params::sockaddr_cb_flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCT_EP_PARAM_FIELD_SOCKADDR_CB_FLAGS</td>
<td>Enables uct_ep_params::sockaddr_pack_cb</td>
</tr>
<tr>
<td>UCT_EP_PARAM_FIELD_SOCKADDR_PACKET_CB</td>
<td>Enables uct_ep_params::cm</td>
</tr>
<tr>
<td>UCT_EP_PARAM_FIELD_CONN_REQUEST</td>
<td>Enables uct_ep_params::conn_request</td>
</tr>
<tr>
<td>UCT_EP_PARAM_FIELD_SOCKADDR_CONNECT_CB_CLIENT</td>
<td>Enables uct_ep_params::sockaddr_client</td>
</tr>
<tr>
<td>UCT_EP_PARAM_FIELD_SOCKADDR_NOTIFY_CB_SERVER</td>
<td>Enables uct_ep_params::sockaddr_server</td>
</tr>
<tr>
<td>UCT_EP_PARAM_FIELD_SOCKADDR_DISCONNECT_CB</td>
<td>Enables uct_ep_params::disconnect_cb</td>
</tr>
<tr>
<td>UCT_EP_PARAM_FIELD_PATH_INDEX</td>
<td>Enables uct_ep_params::path_index</td>
</tr>
</tbody>
</table>

6.11.4.11 anonymous enum

anonymous enum

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Enables uct_ep_params::sockaddr_cb_client</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCT_TAG_RECV_CB_INLINE_DATA</td>
<td>Enables uct_ep_params::sockaddr_cb_server</td>
</tr>
</tbody>
</table>

6.11.4.12 uct_cb_param_flags

enum uct_cb_param_flags

If UCT_CB_PARAM_FLAG_DESC flag is enabled, then data is part of a descriptor which includes the user-defined rx_headroom, and the callback may return UCS_INPROGRESS and hold on to that descriptor. Otherwise, the data can’t be used outside the callback. If needed, the data must be copied-out.

<table>
<thead>
<tr>
<th>descriptor</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>+-----------+------------------</td>
<td></td>
</tr>
<tr>
<td>rx_headroom</td>
<td>payload</td>
</tr>
<tr>
<td>+-----------+------------------</td>
<td></td>
</tr>
</tbody>
</table>

UCT_CB_PARAM_FLAG_FIRST and UCT_CB_PARAM_FLAG_MORE flags are relevant for uct_tag_unexp_eager_cb_t callback only. The former value indicates that the data is the first fragment of the message. The latter value means that more fragments of the message yet to be delivered.

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Enables uct_ep_params::sockaddr_client</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCT_CB_PARAM_FLAG_DESC</td>
<td>Enables uct_ep_params::sockaddr_server</td>
</tr>
<tr>
<td>UCT_CB_PARAM_FLAG_FIRST</td>
<td>Enables uct_ep_params::disconnect_cb</td>
</tr>
</tbody>
</table>

6.11.5 Function Documentation

© 2021 Unified Communication X (UCX). All rights reserved.
6.11.5.1 uct_query_components()

```c
ucss_status_t uct_query_components {
    uct_component_h ** components_p,
    unsigned * num_components_p
}
```

Obtain the list of transport components available on the current system.

Parameters

| out | components_p | Filled with a pointer to an array of component handles. |
| out | num_components_p | Filled with the number of elements in the array. |

Returns

UCS_OK if successful, or UCS_ERR_NO_MEMORY if failed to allocate the array of component handles.

Examples

uct_hello_world.c.

6.11.5.2 uct_release_component_list()

```c
void uct_release_component_list ( 
    uct_component_h * components 
)
```

This routine releases the memory associated with the list of components allocated by uct_query_components.

Parameters

| in | components | Array of component handles to release. |

Examples

uct_hello_world.c.

6.11.5.3 uct_component_query()

```c
ucss_status_t uct_component_query ( 
    uct_component_h component,
    uct_component_attr_t * component_attr 
)
```

Query various attributes of a component.

Parameters

| in | component | Component handle to query attributes for. The handle can be obtained from uct_query_components. |
| in,out | component_attr | Filled with component attributes. |
Returns

UCS_OK if successful, or nonzero error code in case of failure.

Examples

uct_hello_world.c.

6.11.5.4 uct_md_open()

```c
ucs_status_t uct_md_open (
    uct_component_h component,
    const char * md_name,
    const uct_md_config_t * config,
    uct_md_h * md_p )
```

Open a specific memory domain. All communications and memory operations are performed in the context of a specific memory domain. Therefore it must be created before communication resources.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>component</th>
<th>Component on which to open the memory domain, as returned from uct_query_components.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>md_name</td>
<td>Memory domain name, as returned from uct_component_query.</td>
</tr>
<tr>
<td>in</td>
<td>config</td>
<td>MD configuration options. Should be obtained from uct_md_config_read() function, or point to MD-specific structure which extends uct_md_config_t.</td>
</tr>
<tr>
<td>out</td>
<td>md_p</td>
<td>Filled with a handle to the memory domain.</td>
</tr>
</tbody>
</table>

Returns

Error code.

Examples

uct_hello_world.c.

6.11.5.5 uct_md_close()

```c
void uct_md_close ( uct_md_h md )
```

Parameters

| in  | md | Memory domain to close. |

Examples

uct_hello_world.c.
6.11.5.6  uct_md_query_tl_resources()

```c
ucs_status_t uct_md_query_tl_resources (
    uct_md_h md,
    uct_tl_resource_desc_t ** resources_p,
    unsigned * num_resources_p
)
```

This routine queries the memory domain for communication resources that are available for it.

**Parameters**

<table>
<thead>
<tr>
<th>in</th>
<th>md</th>
<th>Handle to memory domain.</th>
</tr>
</thead>
<tbody>
<tr>
<td>out</td>
<td>resources_p</td>
<td>Filled with a pointer to an array of resource descriptors.</td>
</tr>
<tr>
<td>out</td>
<td>num_resources_p</td>
<td>Filled with the number of resources in the array.</td>
</tr>
</tbody>
</table>

**Returns**

Error code.

**Examples**

uct_hello_world.c.

6.11.5.7  uct_release_tl_resource_list()

```c
void uct_release_tl_resource_list ( 
    uct_tl_resource_desc_t * resources 
)
```

This routine releases the memory associated with the list of resources allocated by `uct_md_query_tl_resources`.

**Parameters**

| in  | resources | Array of resource descriptors to release. |

**Examples**

uct_hello_world.c.

6.11.5.8  uct_md_iface_config_read()

```c
ucs_status_t uct_md_iface_config_read ( 
    uct_md_h md,
    const char * tl_name,
    const char * env_prefix,
    const char * filename,
    uct_iface_config_t ** config_p
)
```

**Parameters**

| in  | md | Memory domain on which the transport's interface was registered. |

# Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tl_name</code></td>
<td>Transport name. If <code>md</code> supports <code>UCT_MD_FLAG_SOCKADDR</code>, the transport name is allowed to be NULL. In this case, the configuration returned from this routine should be passed to <code>uct_iface_open</code> with <code>UCT_IFACE_OPEN_MODE_SOCKADDR_SERVER</code> or <code>UCT_IFACE_OPEN_MODE_SOCKADDR_CLIENT</code> set in <code>uct_iface_params_t::open_mode</code>. In addition, if <code>tl_name</code> is not NULL, the configuration returned from this routine should be passed to <code>uct_iface_open</code> with <code>UCT_IFACE_OPEN_MODE_DEVICE</code> set in <code>uct_iface_params_t::open_mode</code>.</td>
</tr>
<tr>
<td><code>env_prefix</code></td>
<td>If non-NULL, search for environment variables starting with this <code>UCT_&lt;prefix&gt;</code>. Otherwise, search for environment variables starting with just <code>UCT_</code>.</td>
</tr>
<tr>
<td><code>filename</code></td>
<td>If non-NULL, read configuration from this file. If the file does not exist, it will be ignored.</td>
</tr>
<tr>
<td><code>config_p</code></td>
<td>Filled with a pointer to configuration.</td>
</tr>
</tbody>
</table>

## Returns

Error code.

## Examples

`uct_hello_world.c`.

## 6.11.5.9 uct_config_release()

```c
void uct_config_release ( void * config )
```

## Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>config</code></td>
<td>Configuration to release.</td>
</tr>
</tbody>
</table>

## Examples

`uct_hello_world.c`.

## 6.11.5.10 uct_iface_open()

```c
ucs_status_t uct_iface_open (  
    uct_md_h md,  
    uct_worker_h worker,  
    const uct_iface_params_t * params,  
    const uct_iface_config_t * config,  
    uct_iface_h * iface_p )
```

## Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>md</code></td>
<td>Memory domain to create the interface on.</td>
</tr>
<tr>
<td><code>worker</code></td>
<td>Handle to worker which will be used to progress communications on this interface.</td>
</tr>
<tr>
<td><code>params</code></td>
<td>User defined <code>uct_iface_params_t</code> parameters.</td>
</tr>
<tr>
<td><code>config</code></td>
<td>Interface configuration options. Should be obtained from <code>uct_md_iface_config_read()</code> function, or point to transport-specific structure which extends <code>uct_iface_config_t</code>.</td>
</tr>
</tbody>
</table>
Parameters

| out | iface__p | Filled with a handle to opened communication interface. |

Returns

Error code.

Examples

uct_hello_world.c.

6.11.5.11 uct_iface_close()

```c
void uct_iface_close ( 
    uct_iface_h iface 
)
```

Parameters

| in | iface | Interface to close. |

Examples

uct_hello_world.c.

6.11.5.12 uct_iface_query()

```c
ucs_status_t uct_iface_query ( 
    uct_iface_h iface, 
    uct_iface_attr_t * iface_attr 
)
```

Parameters

| in | iface | Interface to query. |
| out | iface_attr | Filled with interface attributes. |

Examples

uct_hello_world.c.

6.11.5.13 uct_iface_get_device_address()

```c
ucs_status_t uct_iface_get_device_address ( 
    uct_iface_h iface, 
    uct_device_addr_t * addr 
)
```

Get underlying device address of the interface. All interfaces using the same device would return the same address.
Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>iface</th>
<th>Interface to query.</th>
</tr>
</thead>
<tbody>
<tr>
<td>out</td>
<td>addr</td>
<td>Filled with device address. The size of the buffer provided must be at least uct_iface_attr_t::device_addr_len.</td>
</tr>
</tbody>
</table>

Examples

uct_hello_world.c.

6.11.5.14 uct_iface_get_address()

```c
ucs_status_t uct_iface_get_address (
    uct_iface_h iface,
    uct_iface_addr_t *addr )
```

requires UCT_IFACE_FLAG_CONNECT_TO_IFACE.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>iface</th>
<th>Interface to query.</th>
</tr>
</thead>
<tbody>
<tr>
<td>out</td>
<td>addr</td>
<td>Filled with interface address. The size of the buffer provided must be at least uct_iface_attr_t::iface_addr_len.</td>
</tr>
</tbody>
</table>

Examples

uct_hello_world.c.

6.11.5.15 uct_iface_is_reachable()

```c
int uct_iface_is_reachable ( 
    const uct_iface_h iface,
    const uct_device_addr_t *dev_addr,
    const uct_iface_addr_t *iface_addr )
```

This function checks if a remote address can be reached from a local interface. If the function returns true, it does not necessarily mean a connection and/or data transfer would succeed, since the reachability check is a local operation it does not detect issues such as network mis-configuration or lack of connectivity.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>iface</th>
<th>Interface to check reachability from.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>dev_addr</td>
<td>Device address to check reachability to. It is NULL if iface_attr.dev_addr_len == 0, and must be non-NULL otherwise.</td>
</tr>
<tr>
<td>in</td>
<td>iface_addr</td>
<td>Interface address to check reachability to. It is NULL if iface_attr.iface_addr_len == 0, and must be non-NULL otherwise.</td>
</tr>
</tbody>
</table>
Module Documentation

Returns
Nonzero if reachable, 0 if not.

Examples
uct_hello_world.c.

6.11.5.16 uct_ep_check()

```c
ucs_status_t uct_ep_check (const uct_ep_h ep, unsigned flags, uct_completion_t * comp)
```

This function checks if the destination endpoint is alive with respect to the UCT library. If the status of `ep` is known, either `UCS_OK` or an error is returned immediately. Otherwise, `UCS_INPROGRESS` is returned, indicating that synchronization on the status is needed. In this case, the status will be propagated by `comp` callback.

Parameters

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ep</code></td>
<td>Endpoint to check</td>
</tr>
<tr>
<td><code>flags</code></td>
<td>Flags that define level of check (currently unsupported - set to 0).</td>
</tr>
<tr>
<td><code>comp</code></td>
<td>Handler to process status of <code>ep</code></td>
</tr>
</tbody>
</table>

Returns
Error code.

6.11.5.17 uct_iface_event_fd_get()

```c
ucs_status_t uct_iface_event_fd_get (uct_iface_h iface, int * fd_p)
```

Only interfaces that support at least one of the `UCT_IFACE_FLAG_EVENT*` flags will implement this function.

Parameters

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>iface</code></td>
<td>Interface to get the notification descriptor.</td>
</tr>
<tr>
<td><code>fd_p</code></td>
<td>Location to write the notification file descriptor.</td>
</tr>
</tbody>
</table>

Returns
Error code.

6.11.5.18 uct_iface_event_arm()

```c
ucs_status_t uct_iface_event_arm (```````````````````
```
This routine needs to be called before waiting on each notification on this interface, so will typically be called once the processing of the previous event is over.

Parameters

| in   | iface   | Interface to arm. |
|      | events  | Events to wakeup on. See uct_iface_event_types |

Returns

- UCS_OK The operation completed successfully. File descriptor will be signaled by new events.
- UCS_ERR_BUSY There are unprocessed events which prevent the file descriptor from being armed. The operation is not completed. File descriptor will not be signaled by new events.
- Other different error codes in case of issues.

6.11.5.19 uct_iface_mem_alloc()

```c
ucs_status_t uct_iface_mem_alloc (  
    uct_iface_h iface,  
    size_t length,  
    unsigned flags,  
    const char * name,  
    uct_allocated_memory_t * mem )
```

Allocate a region of memory which can be used for zero-copy data transfer or remote access on a particular transport interface.

Parameters

| in   | iface       | Interface to allocate memory on. |
|      | length      | Size of memory region to allocate. |
|      | flags       | Memory allocation flags, see uct_md_mem_flags. |
| in   | name        | Allocation name, for debug purposes. |
| out  | mem         | Descriptor of allocated memory. |

Returns

UCS_OK if allocation was successful, error code otherwise.

6.11.5.20 uct_iface_mem_free()

```c
void uct_iface_mem_free (  
    const uct_allocated_memory_t * mem )
```

Parameters

| in   | mem       | Descriptor of memory to release. |
### 6.11.5.21 uct_ep_create()

```c
ucs_status_t uct_ep_create (
    const uct_ep_params_t * params,
    uct_ep_h * ep_p )
```

Create a UCT endpoint in one of the available modes:

1. **Unconnected endpoint**: If no any address is present in `uct_ep_params`, this creates an unconnected endpoint. To establish a connection to a remote endpoint, `uct_ep_connect_to_ep` will need to be called. Use of this mode requires `uct_ep_params_t::iface` has the `UCT_IFACE_FLAG_CONNECT_TO_EP` capability flag. It may be obtained by `uct_iface_query`.

2. **Connect to a remote interface**: If `uct_ep_params_t::dev_addr` and `uct_ep_params_t::iface_addr` are set, this will establish an endpoint that is connected to a remote interface. This requires that `uct_ep_params_t::iface` has the `UCT_IFACE_FLAG_CONNECT_TO_IFACE` capability flag. It may be obtained by `uct_iface_query`.

3. **Connect to a remote socket address**: If `uct_ep_params_t::sockaddr` is set, this will create an endpoint that is connected to a remote socket. This requires that either `uct_ep_params::cm`, or `uct_ep_params::iface` will be set. In the latter case, the interface has to support `UCT_IFACE_FLAG_CONNECT_TO_SOCKADDR` flag, which can be checked by calling `uct_iface_query`.

**Parameters**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td><code>params</code> User defined <code>uct_ep_params_t</code> configuration for the <code>ep_p</code>.</td>
</tr>
<tr>
<td>out</td>
<td><code>ep_p</code> Filled with handle to the new endpoint.</td>
</tr>
</tbody>
</table>

**Returns**

- **UCS_OK** The endpoint is created successfully. This does not guarantee that the endpoint has been connected to the destination defined in `params`; in case of failure, the error will be reported to the interface error handler callback provided to `uct_iface_open` via `uct_iface_params_t::err_handler`. Error code as defined by `ucs_status_t`.

**Examples**

- `uct_hello_world.c`

---

### 6.11.5.22 uct_ep_destroy()

```c
void uct_ep_destroy (
    uct_ep_h ep )
```

**Parameters**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td><code>ep</code> Endpoint to destroy.</td>
</tr>
</tbody>
</table>

**Examples**

- `uct_hello_world.c`
6.11.5.23 uct_ep_get_address()

```c
ucs_status_t uct_ep_get_address (  
    uct_ep_h ep,  
    uct_ep_addr_t *addr )
```

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>ep</th>
<th>Endpoint to query.</th>
</tr>
</thead>
<tbody>
<tr>
<td>out</td>
<td>addr</td>
<td>Filled with endpoint address. The size of the buffer provided must be at least <code>uct_iface_attr_t::ep_addr_len</code>.</td>
</tr>
</tbody>
</table>

Examples

`uct_hello_world.c`.

6.11.5.24 uct_ep_connect_to_ep()

```c
ucs_status_t uct_ep_connect_to_ep (  
    uct_ep_h ep,  
    const uct_device_addr_t *dev_addr,  
    const uct_ep_addr_t *ep_addr )
```

requires UCT_IFACE_FLAG_CONNECT_TO_EP capability.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>ep</th>
<th>Endpoint to connect.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>dev_addr</td>
<td>Remote device address.</td>
</tr>
<tr>
<td>in</td>
<td>ep_addr</td>
<td>Remote endpoint address.</td>
</tr>
</tbody>
</table>

Examples

`uct_hello_world.c`.

6.11.5.25 uct_iface_flush()

```c
ucs_status_t uct_iface_flush (  
    uct_iface_h iface,  
    unsigned flags,  
    uct_completion_t *comp )
```

Flushes all outstanding communications issued on the interface prior to this call. The operations are completed at the origin or at the target as well. The exact completion semantic depends on `flags` parameter.

Note

Currently only one completion type is supported. It guarantees that the data transfer is completed but the target buffer may not be updated yet.
Parameters

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>iface</td>
<td>Interface to flush communications from.</td>
</tr>
<tr>
<td>in</td>
<td>flags</td>
<td>Flags that control completion semantic (currently only UCT_FLUSH_FLAG_LOCAL is supported).</td>
</tr>
<tr>
<td>in, out</td>
<td>comp</td>
<td>Completion handle as defined by uct_completion_t. Can be NULL, which means that the call will return the current state of the interface and no completion will be generated in case of outstanding communications. If it is not NULL completion counter is decremented by 1 when the call completes. Completion callback is called when the counter reaches 0.</td>
</tr>
</tbody>
</table>

Returns

UCS_OK - No outstanding communications left. UCS_INPROGRESS - Some communication operations are still in progress. If non-NULL 'comp' is provided, it will be updated upon completion of these operations.

6.11.5.26 uct_iface_fence()

```c
uct_status_t uct_iface_fence (  
    uct_iface_h iface,
    unsigned flags )
```

Parameters

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>iface</td>
<td>Interface to issue communications from.</td>
</tr>
<tr>
<td>in</td>
<td>flags</td>
<td>Flags that control ordering semantic (currently unsupported - set to 0).</td>
</tr>
</tbody>
</table>

Returns

UCS_OK - Ordering is inserted.

6.11.5.27 uct_ep_pending_add()

```c
uct_status_t uct_ep_pending_add (  
    uct_ep_h ep,
    uct_pending_req_t * req,
    unsigned flags )
```

Add a pending request to the endpoint pending queue. The request will be dispatched when the endpoint could potentially have additional send resources.

Parameters

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>ep</td>
<td>Endpoint to add the pending request to.</td>
</tr>
<tr>
<td>in</td>
<td>req</td>
<td>Pending request, which would be dispatched when more resources become available. The user is expected to initialize the &quot;func&quot; field. After being passed to the function, the request is owned by UCT, until the callback is called and returns UCS_OK.</td>
</tr>
<tr>
<td>in</td>
<td>flags</td>
<td>Flags that control pending request processing (see uct_cb_flags)</td>
</tr>
</tbody>
</table>
Returns

UCS_OK - request added to pending queue
UCS_ERR_BUSY - request was not added to pending queue, because send resources are available now. The user is advised to retry.

6.11.5.28 uct_ep_pending_purge()

```c
void uct_ep_pending_purge (
    uct_ep_h ep,
    uct_pending_purge_callback_t cb,
    void * arg)
```

Remove pending requests from the given endpoint and pass them to the provided callback function. The callback return value is ignored.

Parameters

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ep</code></td>
<td>Endpoint to remove pending requests from.</td>
</tr>
<tr>
<td><code>cb</code></td>
<td>Callback to pass the removed requests to.</td>
</tr>
<tr>
<td><code>arg</code></td>
<td>Argument to pass to the <code>cb</code> callback.</td>
</tr>
</tbody>
</table>

6.11.5.29 uct_ep_flush()

```c
ucs_status_t uct_ep_flush (
    uct_ep_h ep,
    unsigned flags,
    uct_completion_t * comp)
```

Flushes all outstanding communications issued on the endpoint prior to this call. The operations are completed at the origin or at the target as well. The exact completion semantic depends on `flags` parameter.

Parameters

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ep</code></td>
<td>Endpoint to flush communications from.</td>
</tr>
<tr>
<td><code>flags</code></td>
<td>Flags <code>uct_flush_flags</code> that control completion semantic.</td>
</tr>
<tr>
<td><code>comp</code></td>
<td>Completion handle as defined by <code>uct_completion_t</code>. Can be NULL, which means that the call will return the current state of the endpoint and no completion will be generated in case of outstanding communications. If it is not NULL completion counter is decremented by 1 when the call completes. Completion callback is called when the counter reaches 0.</td>
</tr>
</tbody>
</table>

Returns

UCS_OK - No outstanding communications left. UCS_ERR_NO_RESOURCE - Flush operation could not be initiated. A subsequent call to `uct_ep_pending_add` would add a pending operation, which provides an opportunity to retry the flush. UCS_INPROGRESS - Some communication operations are still in progress. If non-NULL `comp` is provided, it will be updated upon completion of these operations.

6.11.5.30 uct_ep_fence()

```c
ucs_status_t uct_ep_fence (```

© 2021 Unified Communication X (UCX). All rights reserved.
**uct_ep.h ep, unsigned flags)**

**Parameters**

<table>
<thead>
<tr>
<th>in</th>
<th>ep</th>
<th>Endpoint to issue communications from.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>flags</td>
<td>Flags that control ordering semantic (currently unsupported - set to 0).</td>
</tr>
</tbody>
</table>

**Returns**

UCS_OK - Ordering is inserted.

---

### 6.11.5.31 uct_iface_progress_enable()

```c
void uct_iface_progress_enable (    uct_iface_h iface,    unsigned flags)
```

Notify the transport that it should actively progress communications during `uct_worker_progress()`.

When the interface is created, its progress is initially disabled.

**Parameters**

<table>
<thead>
<tr>
<th>in</th>
<th>iface</th>
<th>The interface to enable progress.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>flags</td>
<td>The type of progress to enable as defined by <code>uct_progress_types</code>.</td>
</tr>
</tbody>
</table>

**Note**

This function is not thread safe with respect to `ucp_worker_progress()`, unless the flag `UCT_PROGRESS_THREAD_SAFE` is specified.

**Examples**

`uct_hello_world.c`

---

### 6.11.5.32 uct_iface_progress_disable()

```c
void uct_iface_progress_disable (    uct_iface_h iface,    unsigned flags)
```

Notify the transport that it should not progress its communications during `uct_worker_progress()`. Thus the latency of other transports may be improved.

By default, progress is disabled when the interface is created.

**Parameters**

<table>
<thead>
<tr>
<th>in</th>
<th>iface</th>
<th>The interface to disable progress.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>flags</td>
<td>The type of progress to disable as defined by <code>uct_progress_types</code>.</td>
</tr>
</tbody>
</table>
Note

This function is not thread safe with respect to \texttt{ucp_worker_progress()}, unless the flag \texttt{UCT\_PROGRESS\_THREAD\_SAFE} is specified.

6.11.5.33 \texttt{uct\_iface\_progress()}

\begin{verbatim}
unsigned uct_iface_progress ( 
    uct_iface_h iface )
\end{verbatim}

6.11.5.34 \texttt{uct\_completion\_update\_status()}

\begin{verbatim}
static UCS\_F\_ALWAYS\_INLINE void uct_completion_update_status ( 
    uct_completion_t * comp, 
    ucs_status_t status ) [static]
\end{verbatim}

**Parameters**

<table>
<thead>
<tr>
<th>comp</th>
<th>[in] Completion handle to update.</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>[in] Status to update \texttt{comp} handle.</td>
</tr>
</tbody>
</table>
6.12 UCT Communication Context

Enumerations

- enum uct_alloc_method_t {
  UCT_ALLOC_METHOD_THP, UCT_ALLOC_METHOD_MD, UCT_ALLOC_METHOD_HEAP, UCT_ALLOC_METHOD_MMAP,
  UCT_ALLOC_METHOD_HUGE, UCT_ALLOC_METHOD_LAST, UCT_ALLOC_METHOD_DEFAULT = UCT_ALLOC_METHOD_LAST
}

Memory allocation methods.

Functions

- ucs_status_t uct_worker_create (ucs_async_context_t *async, ucs_thread_mode_t thread_mode,
  uct_worker_h *worker_p)
  Create a worker object.
- void uct_worker_destroy (uct_worker_h worker)
  Destroy a worker object.
- void uct_worker_progress_register_safe (uct_worker_h worker, ucs_callback_t func, void *arg,
  unsigned flags, uct_worker_cb_id_t *id_p)
  Add a slow path callback function to a worker progress.
- void uct_worker_progress_unregister_safe (uct_worker_h worker, uct_worker_cb_id_t *id_p)
  Remove a slow path callback function from worker's progress.
- ucs_status_t uct_config_get (void *config, const char *name, char *value, size_t max)
  Get value by name from interface configuration (uct_iface_config_t), memory domain configuration (uct_md_config_t)
  or connection manager configuration (uct_cm_config_t).
- ucs_status_t uct_config_modify (void *config, const char *name, const char *value)
  Modify interface configuration (uct_iface_config_t), memory domain configuration (uct_md_config_t) or connection
  manager configuration (uct_cm_config_t).
- unsigned uct_worker_progress (uct_worker_h worker)
  Explicit progress for UCT worker.

6.12.1 Detailed Description

UCT context abstracts all the resources required for network communication. It is designed to enable either share
or isolate resources for multiple programming models used by an application.

This section provides a detailed description of this concept and routines associated with it.

6.12.2 Enumeration Type Documentation

6.12.2.1 uct_alloc_method_t

enum uct_alloc_method_t

Enum

<table>
<thead>
<tr>
<th>UCT_ALLOC_METHOD_THP</th>
<th>Allocate from OS using libc allocator with Transparent Huge Pages enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCT_ALLOC_METHOD_MD</td>
<td>Allocate using memory domain</td>
</tr>
<tr>
<td>UCT_ALLOC_METHOD_HEAP</td>
<td>Allocate from heap using libc allocator</td>
</tr>
<tr>
<td>UCT_ALLOC_METHOD_MMAP</td>
<td>Allocate from OS using mmap() syscall</td>
</tr>
</tbody>
</table>
6.12 UCT Communication Context

### Enumerator

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCT_ALLOC_METHOD_HUGE</td>
<td>Allocate huge pages</td>
</tr>
<tr>
<td>UCT_ALLOC_METHOD_LAST</td>
<td></td>
</tr>
<tr>
<td>UCT_ALLOC_METHOD_DEFAULT</td>
<td>Use default method</td>
</tr>
</tbody>
</table>

#### 6.12.3 Function Documentation

#### 6.12.3.1 uct_worker_create()

```c
ucs_status_t uct_worker_create (  
    ucs_async_context_t *async,  
    ucs_thread_mode_t thread_mode,  
    uct_worker_h *worker_p)
```

The worker represents a progress engine. Multiple progress engines can be created in an application, for example to be used by multiple threads. Transports can allocate separate communication resources for every worker, so that every worker can be progressed independently of others.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in async</td>
<td>Context for async event handlers. Must not be NULL.</td>
</tr>
<tr>
<td>in thread_mode</td>
<td>Thread access mode to the worker and all interfaces and endpoints associated with it.</td>
</tr>
<tr>
<td>out worker_p</td>
<td>Filled with a pointer to the worker object.</td>
</tr>
</tbody>
</table>

**Examples**

    uct_hello_world.c.

#### 6.12.3.2 uct_worker_destroy()

```c
void uct_worker_destroy (  
    uct_worker_h worker)
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in worker</td>
<td>Worker object to destroy.</td>
</tr>
</tbody>
</table>

**Examples**

    uct_hello_world.c.

#### 6.12.3.3 uct_worker_progress_register_safe()

```c
void uct_worker_progress_register_safe (  
    uct_worker_h worker,
```

© 2021 Unified Communication X (UCX). All rights reserved.
If *id_p is equal to UCS_CALLBACKQ_ID_NULL, this function will add a callback which will be invoked every time progress is made on the worker. *id_p will be updated with an id which refers to this callback and can be used in uct_worker_progress_unregister_safe to remove it from the progress path.

Parameters

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>worker</td>
<td>Handle to the worker whose progress should invoke the callback.</td>
</tr>
<tr>
<td>in</td>
<td>func</td>
<td>Pointer to the callback function.</td>
</tr>
<tr>
<td>in</td>
<td>arg</td>
<td>Argument for the callback function.</td>
</tr>
<tr>
<td>in,out</td>
<td>flags</td>
<td>Callback flags, see ucs_callbackq_flags.</td>
</tr>
<tr>
<td>in,out</td>
<td>id_p</td>
<td>Points to a location to store a callback identifier. If *id_p is equal to UCS_CALLBACKQ_ID_NULL, a callback will be added and *id_p will be replaced with a callback identifier which can be subsequently used to remove the callback. Otherwise, no callback will be added and *id_p will be left unchanged.</td>
</tr>
</tbody>
</table>

Note

This function is thread safe.

6.12.3.4 uct_worker_progress_unregister_safe()

void uct_worker_progress_unregister_safe (  
    uct_worker_h worker,  
    uct_worker_cb_id_t * id_p )

If *id_p is not equal to UCS_CALLBACKQ_ID_NULL, remove a callback which was previously added by uct_worker_progress_register_safe. *id_p will be reset to UCS_CALLBACKQ_ID_NULL.

Parameters

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>worker</td>
<td>Handle to the worker whose progress should invoke the callback.</td>
</tr>
<tr>
<td>in,out</td>
<td>id_p</td>
<td>Points to a callback identifier which indicates the callback to remove. If *id_p is not equal to UCS_CALLBACKQ_ID_NULL, the callback will be removed and *id_p will be reset to UCS_CALLBACKQ_ID_NULL. If *id_p is equal to UCS_CALLBACKQ_ID_NULL, no operation will be performed and *id_p will be left unchanged.</td>
</tr>
</tbody>
</table>

Note

This function is thread safe.

6.12.3.5 uct_config_get()

ucs_status_t uct_config_get (  
    void * config,  
    const char * name,
`char * value,`  
`size_t max`)

**Parameters**

| in | config | Configuration to get from. |
| in | name   | Configuration variable name. |
| out| value  | Pointer to get value. Should be allocated/freed by caller. |
| in | max    | Available memory space at value pointer. |

**Returns**

UCS_OK if found, otherwise UCS_ERR_INVALID_PARAM or UCS_ERR_NO_ELEM if error.

### 6.12.3.6 uct_config_modify()

```c
ucs_status_t uct_config_modify (  
    void * config,  
    const char * name,  
    const char * value)
```

**Parameters**

| in | config | Configuration to modify. |
| in | name   | Configuration variable name. |
| in | value  | Value to set. |

**Returns**

Error code.

### 6.12.3.7 uct_worker_progress()

```c
unsigned uct_worker_progress (  
    uct_worker_h worker)
```

This routine explicitly progresses any outstanding communication operations and active message requests.

**Note**

- In the current implementation, users **MUST** call this routine to receive the active message requests.

**Parameters**

| in | worker | Handle to worker. |
Returns
Nonzero if any communication was progressed, zero otherwise.

Examples
uct_hello_world.c.
6.13 UCT Memory Domain

Data Structures

- struct uct_md_attr
  
  Memory domain attributes. More...

- struct uct_md_attr.cap

- struct uct_md_mem_attr
  
  Memory domain attributes. More...

- struct uct_allocated_memory
  
  Describes a memory allocated by UCT. More...

- struct uct_rkey_bundle
  
  Remote key with its type. More...

- struct uct_mem_alloc_params_t
  
  Parameters for allocating memory using uct_mem_alloc. More...

- struct uct_mem_alloc_params_t.mds

Typedefs

- typedef struct uct_md_mem_attr uct_md_mem_attr_t
  
  Memory domain attributes.

- typedef struct uct_allocated_memory uct_allocated_memory_t
  
  Describes a memory allocated by UCT.

- typedef struct uct_rkey_bundle uct_rkey_bundle_t
  
  Remote key with its type.

Enumerations

- enum uct_sockaddr_accessibility_t { UCT.SOCKADDR_ACC_LOCAL, UCT.SOCKADDR_ACC_REMOTE }
  
  Socket address accessibility type.

- enum {
  UCT_MD_FLAG_ALLOC = UCS_BIT(0), UCT_MD_FLAG_REG = UCS_BIT(1), UCT_MD_FLAG_NEED_MEMH = UCS_BIT(2),
  UCT_MD_FLAG_NEED_RKEY = UCS_BIT(3), UCT_MD_FLAG_ADVISE = UCS_BIT(4), UCT_MD_FLAG_FIXED = UCS_BIT(5),
  UCT_MD_FLAG_RKEY_PTR = UCS_BIT(6), UCT_MD_FLAG_SOCKADDR = UCS_BIT(7) }
  
  Memory domain capability flags.

- enum uct_md_mem_flags {
  UCT_MD_MEM_FLAG_NONBLOCK = UCS_BIT(0), UCT_MD_MEM_FLAG_FIXED = UCS_BIT(1),
  UCT_MD_MEM_FLAG_LOCK = UCS_BIT(2), UCT_MD_MEM_FLAG_HIDE_ERRORS = UCS_BIT(3),
  UCT_MD_MEM_ACCESS_REMOTE_PUT = UCS_BIT(5), UCT_MD_MEM_ACCESS_REMOTE_GET = UCS_BIT(6),
  UCT_MD_MEM_ACCESS_REMOTE_ATOMIC = UCS_BIT(7), UCT_MD_MEM_ACCESS_LOCAL_READ = UCS_BIT(8),
  UCT_MD_MEM_ACCESS_LOCAL_WRITE = UCS_BIT(9), UCT_MD_MEM_ACCESS_ALL, UCT_MD_MEM_ACCESS_RMA }
  
  Memory allocation/registration flags.

- enum uct_mem_advice_t { UCT_MADV_NORMAL = 0, UCT_MADV_WILLNEED }
  
  List of UCT memory use advice.

- enum uct_md_mem_attr_field { UCT_MD_MEM_ATTR_FIELD_MEM_TYPE = UCS_BIT(0), UCT_MD_MEM_ATTR_FIELD_SYS_DEV = UCS_BIT(1) }
  
  UCT MD memory attributes field mask.
enum uct_mem_alloc_params_field_t {
    UCT_MEM_ALLOC_PARAM_FIELD_FLAGS = UCS_BIT(0),
    UCT_MEM_ALLOC_PARAM_FIELD_ADDRESS = UCS_BIT(1),
    UCT_MEM_ALLOC_PARAM_FIELD_MEM_TYPE = UCS_BIT(2),
    UCT_MEM_ALLOC_PARAM_FIELD_MDS = UCS_BIT(3),
    UCT_MEM_ALLOC_PARAM_FIELD_NAME = UCS_BIT(4)
};

UCT allocation parameters specification field mask.

Functions

- ucs_status_t uct_md_mem_query (uct_md_h md, const void *address, const size_t length, uct_md_mem_attr_t *mem_attr)
  Query attributes of a given pointer.
- ucs_status_t uct_md_query (uct_md_h md, uct_md_attr_t *md_attr)
  Query for memory domain attributes.
- ucs_status_t uct_md_mem_advise (uct_md_h md, uct_mem_h memh, void *addr, size_t length, uct_mem_advice_t *advice)
  Give advice about the use of memory.
- ucs_status_t uct_md_mem_reg (uct_md_h md, void *address, size_t length, unsigned flags, uct_mem_h *memh_p)
  Register memory for zero-copy sends and remote access.
- ucs_status_t uct_md_mem_dereg (uct_md_h md, uct_mem_h memh)
  Undo the operation of uct_md_mem_reg().
- ucs_status_t uct_md_detect_memory_type (uct_md_h md, const void *addr, size_t length, ucs_memory_type_t *mem_type_p)
  Detect memory type.
- ucs_status_t uct_md_mem_alloc (size_t length, const uct_alloc_method_t *methods, unsigned num_methods, const uct_mem_alloc_params_t *params, uct_allocated_memory_t *mem)
  Allocate memory for zero-copy communications and remote access.
- ucs_status_t uct_md_mem_free (const uct_allocated_memory_t *mem)
  Release allocated memory.
- ucs_status_t uct_md_config_read (uct_component_h component, const char *env_prefix, const char *filename, uct_md_config_t **config_p)
  Read the configuration for a memory domain.
- int uct_md_is_sockaddr_accessible (uct_md_h md, const ucs_sock_addr_t *sockaddr, uct_sockaddr_accessibility_t mode)
  Check if remote sock address is accessible from the memory domain.
- ucs_status_t uct_md_mkey_pack (uct_md_h md, uct_mem_h memh, void *rkey_buffer)
  Pack a remote key.
- ucs_status_t uct_rkey_unpack (uct_component_h component, const void *rkey_buffer, uct_rkey_bundle_t *rkey_ob)
  Unpack a remote key.
- ucs_status_t uct_rkey_ptr (uct_component_h component, uct_rkey_bundle_t *rkey_ob, uint64_t remote_addr, void **addr_p)
  Get a local pointer to remote memory.
- ucs_status_t uct_rkey_release (uct_component_h component, const uct_rkey_bundle_t *rkey_ob)
  Release a remote key.

6.13.1 Detailed Description

The Memory Domain abstracts resources required for network communication, which typically includes memory, transport mechanisms, compute and network resources. It is an isolation mechanism that can be employed by the applications for isolating resources between multiple programming models. The attributes of the Memory Domain are defined by the structure uct_md_attr(). The communication and memory operations are defined in the context of Memory Domain.
6.13 UCT Memory Domain

6.13.2 Data Structure Documentation

6.13.2.1 struct uct_md_attr

This structure defines the attributes of a Memory Domain which includes maximum memory that can be allocated, credentials required for accessing the memory, CPU mask indicating the proximity of CPUs, and bitmaps indicating the types of memory (CPU/CUDA/ROCM) that can be detected, allocated and accessed.

Examples

uct_hello_world.c.

Data Fields

<table>
<thead>
<tr>
<th>struct uct_md_attr</th>
<th>cap</th>
<th>Maximal allocation size</th>
</tr>
</thead>
<tbody>
<tr>
<td>ucs_linear_func_t</td>
<td>reg_cost</td>
<td>Memory registration cost estimation (time,seconds) as a linear function of the buffer size.</td>
</tr>
<tr>
<td>char</td>
<td>component_name[UCT_COMPONENT_NAME_MAX]</td>
<td>Component name</td>
</tr>
<tr>
<td>size_t</td>
<td>rkeypacked_size</td>
<td>Size of buffer needed for packed rkey</td>
</tr>
<tr>
<td>ucs_cpu_set_t</td>
<td>local_cpus</td>
<td>Mask of CPUs near the resource</td>
</tr>
</tbody>
</table>

6.13.2.2 struct uct_md_attr.cap

Data Fields

<table>
<thead>
<tr>
<th>size_t</th>
<th>max_alloc</th>
<th>Maximal allocation size</th>
</tr>
</thead>
<tbody>
<tr>
<td>size_t</td>
<td>max_reg</td>
<td>Maximal registration size</td>
</tr>
<tr>
<td>uint64_t</td>
<td>flags</td>
<td>UCT_MD_FLAG_xx</td>
</tr>
<tr>
<td>uint64_t</td>
<td>reg_mem_types</td>
<td>Bitmap of memory types that Memory Domain can be registered with</td>
</tr>
<tr>
<td>uint64_t</td>
<td>detect_mem_types</td>
<td>Bitmap of memory types that Memory Domain can detect if address belongs to it</td>
</tr>
<tr>
<td>uint64_t</td>
<td>alloc_mem_types</td>
<td>Bitmap of memory types that Memory Domain can allocate memory on</td>
</tr>
<tr>
<td>uint64_t</td>
<td>access_mem_types</td>
<td>Memory types that Memory Domain can access</td>
</tr>
</tbody>
</table>

6.13.2.3 struct uct_md_mem_attr

This structure defines the attributes of a memory pointer which may include the memory type of the pointer, and the system device that backs the pointer depending on the bit fields populated in field_mask.

Data Fields

<table>
<thead>
<tr>
<th>uint64_t</th>
<th>field_mask</th>
<th>Mask of valid fields in this structure, using bits from uct_md_mem_attr_t. Note that the field mask is populated upon return from uct_md_mem_query and not set by user. Subsequent use of members of the structure are valid after ensuring that relevant bits in the field_mask are set.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ucs_memory_type_t</td>
<td>mem_type</td>
<td>The type of memory. E.g. CPU/GPU memory or some other valid type</td>
</tr>
<tr>
<td>ucs_sys_device_t</td>
<td>sys_dev</td>
<td>Index of the system device on which the buffer resides. E.g: NUMA/GPU</td>
</tr>
</tbody>
</table>
6.13.2.4 struct uct_allocated_memory

This structure describes the memory block which includes the address, size, and Memory Domain used for allocation. This structure is passed to interface and the memory is allocated by memory allocation functions uct_mem_alloc.

Data Fields

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void *</td>
<td>address</td>
<td>Address of allocated memory</td>
</tr>
<tr>
<td>size_t</td>
<td>length</td>
<td>Real size of allocated memory</td>
</tr>
<tr>
<td>uct_alloc_method_t</td>
<td>method</td>
<td>Method used to allocate the memory</td>
</tr>
<tr>
<td>ucs_memory_type_t</td>
<td>mem_type</td>
<td>type of allocated memory</td>
</tr>
<tr>
<td>uct_md_h</td>
<td>md</td>
<td>if method==MD: MD used to allocate the memory</td>
</tr>
<tr>
<td>uct_mem_h</td>
<td>memh</td>
<td>if method==MD: MD memory handle</td>
</tr>
</tbody>
</table>

6.13.2.5 struct uct_rkey_bundle

This structure describes the credentials (typically key) and information required to access the remote memory by the communication interfaces.

Data Fields

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uct_rkey_t</td>
<td>rkey</td>
<td>Remote key descriptor, passed to RMA functions</td>
</tr>
<tr>
<td>void *</td>
<td>handle</td>
<td>Handle, used internally for releasing the key</td>
</tr>
<tr>
<td>void *</td>
<td>type</td>
<td>Remote key type</td>
</tr>
</tbody>
</table>

6.13.2.6 struct uct_mem_alloc_params_t

Data Fields

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint64_t</td>
<td>field_mask</td>
<td>Mask of valid fields in this structure, using bits from uct_mem_alloc_params_field_t. Fields not specified in this mask will be ignored.</td>
</tr>
<tr>
<td>unsigned</td>
<td>flags</td>
<td>Memory allocation flags, see uct_md_mem_flags If UCT_MEM_ALLOC_PARAM_FIELD_FLAGS is not specified in field_mask, then (UCT_MD_MEM_ACCESS_LOCAL_READ</td>
</tr>
<tr>
<td>void *</td>
<td>address</td>
<td>If address is NULL, the underlying allocation routine will choose the address at which to create the mapping. If address is non-NULL and UCT_MD_MEM_FLAG_FIXED is not set, the address will be interpreted as a hint as to where to establish the mapping. If address is non-NULL and UCT_MD_MEM_FLAG_FIXED is set, then the specified address is interpreted as a requirement. In this case, if the mapping to the exact address cannot be made, the allocation request fails.</td>
</tr>
<tr>
<td>ucs_memory_type_t</td>
<td>mem_type</td>
<td>Type of memory to be allocated.</td>
</tr>
<tr>
<td>struct uct_mem_alloc_params_t</td>
<td>mds</td>
<td></td>
</tr>
</tbody>
</table>

© 2021 Unified Communication X (UCX). All rights reserved.
6.13 UCT Memory Domain

Data Fields

| const char * name | Name of the allocated region, used to track memory usage for debugging and profiling. If UCT_MEM_ALLOC_PARAM_FIELD_NAME is not specified in field_mask, then "anonymous-uct_mem_alloc" is used by default. |

6.13.2.7 struct uct_mem_alloc_params_t.mds

Data Fields

| const uct_md_h * mds | Array of memory domains to attempt to allocate the memory with, for MD allocation method. |
| unsigned count | Length of 'mds' array. May be empty, in such case 'mds' may be NULL, and MD allocation method will be skipped. |

6.13.3 Typedef Documentation

6.13.3.1 uct_md_mem_attr_t
typedef struct uct_md_mem_attr uct_md_mem_attr_t

This structure defines the attributes of a memory pointer which may include the memory type of the pointer, and the system device that backs the pointer depending on the bit fields populated in field_mask.

6.13.3.2 uct_allocated_memory_t
typedef struct uct_allocated_memory uct_allocated_memory_t

This structure describes the memory block which includes the address, size, and Memory Domain used for allocation. This structure is passed to interface and the memory is allocated by memory allocation functions uct_mem_alloc.

6.13.3.3 uct_rkey_bundle_t
typedef struct uct_rkey_bundle uct_rkey_bundle_t

This structure describes the credentials (typically key) and information required to access the remote memory by the communication interfaces.

6.13.4 Enumeration Type Documentation

6.13.4.1 uct_sockaddr_accessibility_t
enum uct_sockaddr_accessibility_t

© 2021 Unified Communication X (UCX). All rights reserved.
### Enumerator

<table>
<thead>
<tr>
<th>UCT_SOCKET_ADDR_ACC_LOCAL</th>
<th>Check if local address exists. Address should belong to a local network interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCT_SOCKET_ADDR_ACC_REMOTE</td>
<td>Check if remote address can be reached. Address is routable from one of the local network interfaces</td>
</tr>
</tbody>
</table>

### 6.13.4.2 anonymous enum

**anonymous enum**

<table>
<thead>
<tr>
<th>Enumerator</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCT_MD_FLAG_ALLOC</td>
</tr>
<tr>
<td>UCT_MD_FLAG_REG</td>
</tr>
<tr>
<td>UCT_MD_FLAG_NEED_MEMH</td>
</tr>
<tr>
<td>UCT_MD_FLAG_NEED_RKEY</td>
</tr>
<tr>
<td>UCT_MD_FLAG_ADVISE</td>
</tr>
<tr>
<td>UCT_MD_FLAG_FIXED</td>
</tr>
<tr>
<td>UCT_MD_FLAG_RKEY_PTR</td>
</tr>
<tr>
<td>UCT_MD_FLAG_SOCKETADDR</td>
</tr>
</tbody>
</table>

### 6.13.4.3 uct_md_mem_flags

**enum uct_md_mem_flags**

<table>
<thead>
<tr>
<th>Enumerator</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCT_MD_MEM_FLAG_NONBLOCK</td>
</tr>
<tr>
<td>UCT_MD_MEM_FLAG_FIXED</td>
</tr>
<tr>
<td>UCT_MD_MEM_FLAG_LOCK</td>
</tr>
<tr>
<td>UCT_MD_MEM_FLAG_HIDE_ERRORS</td>
</tr>
<tr>
<td>UCT_MD_MEM_ACCESS_REMOTE_PUT</td>
</tr>
<tr>
<td>UCT_MD_MEM_ACCESS_REMOTE_GET</td>
</tr>
<tr>
<td>UCT_MD_MEM_ACCESS_REMOTE_ATOMIC</td>
</tr>
<tr>
<td>UCT_MD_MEM_ACCESS_LOCAL_READ</td>
</tr>
<tr>
<td>UCT_MD_MEM_ACCESS_LOCAL_WRITE</td>
</tr>
<tr>
<td>UCT_MD_MEM_ACCESS_ALL</td>
</tr>
<tr>
<td>UCT_MD_MEM_ACCESS_RMA</td>
</tr>
</tbody>
</table>
6.13.4.4 uct_mem_advice_t

enum uct_mem_advice_t

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCT_MADV_NORMAL</td>
<td>No special treatment</td>
</tr>
<tr>
<td>UCT_MADV_WILLNEED</td>
<td>can be used on the memory mapped with UCT_MD_MEM_FLAG_NONBLOCK to speed up memory mapping and to avoid page faults when the memory is accessed for the first time.</td>
</tr>
</tbody>
</table>

6.13.4.5 uct_md_mem_attr_field

enum uct_md_mem_attr_field

The enumeration allows specifying which fields in uct_md_mem_attr_t are present.

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCT_MD_MEM_ATTR_FIELD_MEM_TYPE</td>
<td>Indicate if memory type is populated. E.g. CPU/GPU</td>
</tr>
<tr>
<td>UCT_MD_MEM_ATTR_FIELD_SYS_DEV</td>
<td>Indicate if details of system device backing the pointer are populated. E.g. NUMA/GPU</td>
</tr>
</tbody>
</table>

6.13.4.6 uct_mem_alloc_params_field_t

enum uct_mem_alloc_params_field_t

The enumeration allows specifying which fields in uct_mem_alloc_params_t are present.

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCT_MEM_ALLOC_PARAM_FIELD_FLAGS</td>
<td>Enables uct_mem_alloc_params_t::flags</td>
</tr>
<tr>
<td>UCT_MEM_ALLOC_PARAM_FIELD_ADDRESS</td>
<td>Enables uct_mem_alloc_params_t::address</td>
</tr>
<tr>
<td>UCT_MEM_ALLOC_PARAM_FIELD_MEM_TYPE</td>
<td>Enables uct_mem_alloc_params_t::mem_type</td>
</tr>
<tr>
<td>UCT_MEM_ALLOC_PARAM_FIELD_MDS</td>
<td>Enables uct_mem_alloc_params_t::mrs</td>
</tr>
<tr>
<td>UCT_MEM_ALLOC_PARAM_FIELD_NAME</td>
<td>Enables uct_mem_alloc_params_t::name</td>
</tr>
</tbody>
</table>

6.13.5 Function Documentation

6.13.5.1 uct_md_mem_query()

```c
ucs_status_t uct_md_mem_query (        uct_md_h md,
```
const void * address,
const size_t length,
uct_md_mem_attr_t * mem_attr )

Return attributes such as memory type, and system device for the given pointer of specific length.

Parameters

| in | md | Memory domain to run the query on. This function returns an error if the md does not recognize the pointer. |
| in | address | The address of the pointer. Must be non-NULL else UCS_ERR_INVALID_PARAM error is returned. |
| in | length | Length of the memory region to examine. Must be nonzero else UCS_ERR_INVALID_PARAM error is returned. |
| out | mem_attr | If successful, filled with ptr attributes. |

Returns

Error code.

6.13.5.2 uct_md_query()

uct_status_t uct_md_query ( 
    uct_md_h md,
    uct_md_attr_t * md_attr )

Parameters

| in | md | Memory domain to query. |
| out | md_attr | Filled with memory domain attributes. |

Examples

uct_hello_world.c.

6.13.5.3 uct_md_mem_advise()

uct_status_t uct_md_mem_advise ( 
    uct_md_h md,
    uct_mem_h memh,
    void * addr,
    size_t length,
    uct_mem_advice_t advice )

This routine advises the UCT about how to handle memory range beginning at address and size of length bytes. This call does not influence the semantics of the application, but may influence its performance. The advice may be ignored.

Parameters

| in | md | Memory domain memory was allocated or registered on. |
6.13 UCT Memory Domain

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>memh</code></td>
<td>Memory handle, as returned from <code>uct_mem_alloc</code></td>
</tr>
<tr>
<td><code>addr</code></td>
<td>Memory base address. Memory range must belong to the <code>memh</code></td>
</tr>
<tr>
<td><code>length</code></td>
<td>Length of memory to advise. Must be &gt;0.</td>
</tr>
<tr>
<td><code>advice</code></td>
<td>Memory use advice as defined in the <code>uct_mem_advice_t</code> list</td>
</tr>
</tbody>
</table>

6.13.5.4 uct_md_mem_reg()

```c
ucs_status_t uct_md_mem_reg (uct_md_h md, void *address, size_t length, unsigned flags, uct_mem_h *memh_p)
```

Register memory on the memory domain. In order to use this function, MD must support `UCT_MD_FLAG_REG` flag.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>md</code></td>
<td>Memory domain to register memory on.</td>
</tr>
<tr>
<td><code>address</code></td>
<td>Memory to register.</td>
</tr>
<tr>
<td><code>length</code></td>
<td>Size of memory to register. Must be &gt;0.</td>
</tr>
<tr>
<td><code>flags</code></td>
<td>Memory allocation flags, see <code>uct_md_mem_flags</code>.</td>
</tr>
<tr>
<td><code>memh_p</code></td>
<td>Filled with handle for allocated region.</td>
</tr>
</tbody>
</table>

Examples

`uct_hello_world.c`

6.13.5.5 uct_md_mem_dereg()

```c
ucs_status_t uct_md_mem_dereg (uct_md_h md, uct_mem_h memh)
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>md</code></td>
<td>Memory domain which was used to register the memory.</td>
</tr>
<tr>
<td><code>memh</code></td>
<td>Local access key to memory region.</td>
</tr>
</tbody>
</table>

Examples

`uct_hello_world.c`
6.13.5.6 uct_md_detect_memory_type()

```c
ucs_status_t uct_md_detect_memory_type(
    uct_md_h md,
    const void *addr,
    size_t length,
    ucs_memory_type_t *mem_type_p)
```

**Parameters**

<table>
<thead>
<tr>
<th>in</th>
<th>md</th>
<th>Memory domain to detect memory type</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>addr</td>
<td>Memory address to detect.</td>
</tr>
<tr>
<td>in</td>
<td>length</td>
<td>Size of memory</td>
</tr>
<tr>
<td>out</td>
<td>mem_type_p</td>
<td>Filled with memory type of the address range if function succeeds</td>
</tr>
</tbody>
</table>

**Returns**

UCS_OK if memory type is successfully detected UCS_ERR_INVALID_ADDR if failed to detect memory type

6.13.5.7 uct_mem_alloc()

```c
ucs_status_t uct_mem_alloc(
    size_t length,
    const uct_alloc_method_t *methods,
    unsigned num_methods,
    const uct_mem_alloc_params_t *params,
    uct_allocated_memory_t *mem)
```

Allocate potentially registered memory.

**Parameters**

<table>
<thead>
<tr>
<th>in</th>
<th>length</th>
<th>The minimal size to allocate. The actual size may be larger, for example because of alignment restrictions. Must be &gt;0.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>methods</td>
<td>Array of memory allocation methods to attempt. Each of the provided allocation methods will be tried in array order, to perform the allocation, until one succeeds. Whenever the MD method is encountered, each of the provided MDs will be tried in array order, to allocate the memory, until one succeeds, or they are exhausted. In this case the next allocation method from the initial list will be attempted.</td>
</tr>
<tr>
<td>in</td>
<td>num_methods</td>
<td>Length of 'methods' array.</td>
</tr>
<tr>
<td>in</td>
<td>params</td>
<td>Memory allocation characteristics, see uct_mem_alloc_params_t.</td>
</tr>
<tr>
<td>out</td>
<td>mem</td>
<td>In case of success, filled with information about the allocated memory. uct_allocated_memory_t</td>
</tr>
</tbody>
</table>

6.13.5.8 uct_mem_free()

```c
ucs_status_t uct_mem_free(
    const uct_allocated_memory_t *mem)
```

Release the memory allocated by uct_mem_alloc.
6.13 UCT Memory Domain

Parameters

| in | mem | Description of allocated memory, as returned from uct_mem_alloc. |

6.13.5.9 uct_md_config_read()

```c
uct_status_t uct_md_config_read(
    uct_component_h component,
    const char *env_prefix,  
    const char *filename,    
    uct_md_config_t **config_p)
```

Parameters

| in  | component | Read the configuration of this component. |
| in  | env_prefix | If non-NULL, search for environment variables starting with this UCT_<prefix>_. Otherwise, search for environment variables starting with just UCT_. |
| in  | filename  | If non-NULL, read configuration from this file. If the file does not exist, it will be ignored. |
| out | config_p  | Filled with a pointer to the configuration. |

Returns

Error code.

Examples

uct_hello_world.c.

6.13.5.10 uct_md_is_sockaddr_accessible()

```c
int uct_md_is_sockaddr_accessible(
    uct_md_h md,         
    const ucs_sock_addr_t *sockaddr,
    uct_sockaddr_accessibility_t mode)
```

This function checks if a remote sock address can be accessed from a local memory domain. Accessibility can be checked in local or remote mode.

Parameters

| in  | md | Memory domain to check accessibility from. This memory domain must support the UCT_MD_FLAG_SOCKADDR flag. |
| in  | sockaddr | Socket address to check accessibility to. |
| in  | mode | Mode for checking accessibility, as defined in uct_sockaddr_accessibility_t. Indicates if accessibility is tested on the server side - for binding to the given sockaddr, or on the client side - for connecting to the given remote peer's sockaddr. |

© 2021 Unified Communication X (UCX). All rights reserved.
Returns

Nonzero if accessible, 0 if inaccessible.

6.13.5.11 uct_md_mkey_pack()

```c
ucs_status_t uct_md_mkey_pack (  
    uct_md_h md,  
    uct_mem_h memh,  
    void * rkey_buffer)
```

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>md</th>
<th>Handle to memory domain.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>memh</td>
<td>Local key, whose remote key should be packed.</td>
</tr>
<tr>
<td>out</td>
<td>rkey_buffer</td>
<td>Filled with packed remote key.</td>
</tr>
</tbody>
</table>

Returns

Error code.

6.13.5.12 uct_rkey_unpack()

```c
ucs_status_t uct_rkey_unpack (  
    uct_component_h component,  
    const void * rkey_buffer,  
    uct_rkey_bundle_t * rkey_ob)
```

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>component</th>
<th>Component on which to unpack the remote key.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>rkey_buffer</td>
<td>Packed remote key buffer.</td>
</tr>
<tr>
<td>out</td>
<td>rkey_ob</td>
<td>Filled with the unpacked remote key and its type.</td>
</tr>
</tbody>
</table>

Note

The remote key must be unpacked with the same component that was used to pack it. For example, if a remote device address on the remote memory domain which was used to pack the key is reachable by a transport on a local component, then that component is eligible to unpack the key. If the remote key buffer cannot be unpacked with the given component, UCS_ERR_INVALID_PARAM will be returned.

Returns

Error code.

6.13.5.13 uct_rkey_ptr()

```c
ucs_status_t uct_rkey_ptr (  
    uct_component_h component,
```
uct_rkey_bundle_t * rkey_ob,
uint64_t remote_addr,
void ** addr_p)

This routine returns a local pointer to the remote memory described by the rkey bundle. The MD must support UCT_MD_FLAG_RKEY_PTR flag.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>component</th>
<th>Component on which to obtain the pointer to the remote key.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>rkey_ob</td>
<td>A remote key bundle as returned by the uct_rkey_unpack function.</td>
</tr>
<tr>
<td>in</td>
<td>remote_addr</td>
<td>A remote address within the memory area described by the rkey_ob.</td>
</tr>
<tr>
<td>out</td>
<td>addr_p</td>
<td>A pointer that can be used for direct access to the remote memory.</td>
</tr>
</tbody>
</table>

Note

The component used to obtain a local pointer to the remote memory must be the same component that was used to pack the remote key. See notes section for uct_rkey_unpack.

Returns

Error code if the remote memory cannot be accessed directly or the remote address is not valid.

6.13.5.14 uct_rkey_release()

ucs_status_t uct_rkey_release (  
    uct_component_h component,  
    const uct_rkey_bundle_t * rkey_ob )

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>component</th>
<th>Component which was used to unpack the remote key.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>rkey_ob</td>
<td>Remote key to release.</td>
</tr>
</tbody>
</table>
6.14 UCT Active messages

Typedefs

- typedef ucs_status_t(* uct_am_callback_t) (void *arg, void *data, size_t length, unsigned flags)
  Callback to process incoming active message.
- typedef void(* uct_am_tracer_t) (void *arg, uct_am_trace_type_t type, uint8_t id, const void *data, size_t length, char *buffer, size_t max)
  Callback to trace active messages.

Enumerations

- enum uct_msg_flags { UCT_SEND_FLAG_SIGNALED = UCS_BIT(0) }
  Flags for active message send operation.
- enum uct_am_trace_type {
  UCT_AM_TRACE_TYPE_SEND, UCT_AM_TRACE_TYPE_RECV, UCT_AM_TRACE_TYPE_SEND_DROP,
  UCT_AM_TRACE_TYPE_RECV_DROP,
  UCT_AM_TRACE_TYPE_LAST }
  Trace types for active message tracer.

Functions

- ucs_status_t uct_iface_set_am_handler (uct_iface_h iface, uint8_t id, uct_am_callback_t cb, void *arg, uint32_t flags)
  Set active message handler for the interface.
- ucs_status_t uct_iface_set_am_tracer (uct_iface_h iface, uct_am_tracer_t tracer, void *arg)
  Set active message tracer for the interface.
- void uct_iface_release_desc (void *desc)
  Release AM descriptor.
- ucs_status_t uct_ep_am_short (uct_ep_h ep, uint8_t id, uint64_t header, const void *payload, unsigned length)
  Send active message while avoiding local memory copy.
- ssize_t uct_ep_am_bcopy (uct_ep_h ep, uint8_t id, uct_pack_callback_t pack_cb, void *arg, unsigned flags)
- ucs_status_t uct_ep_am_zcopy (uct_ep_h ep, uint8_t id, const void *header, unsigned header_length, const uct_iov_t *iov, size_t iovcnt, unsigned flags, uct_completion_t *comp)
  Send active message while avoiding local memory copy.

6.14.1 Detailed Description

Defines active message functions.

6.14.2 Typedef Documentation

6.14.2.1 uct_am_callback_t

typedef ucs_status_t(* uct_am_callback_t) (void *arg, void *data, size_t length, unsigned flags)

When the callback is called, flags indicates how data should be handled. If flags contain UCT_CB_PARAM_FLAG_DESC value, it means data is part of a descriptor which must be released later by uct_iface_release_desc by the user if the callback returns UCS_INPROGRESS.
Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>arg</th>
<th>User-defined argument.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>data</td>
<td>Points to the received data. This may be a part of a descriptor which may be released later.</td>
</tr>
<tr>
<td>in</td>
<td>length</td>
<td>Length of data.</td>
</tr>
<tr>
<td>in</td>
<td>flags</td>
<td>Mask with uct_cb_param_flags</td>
</tr>
</tbody>
</table>

Note

This callback could be set and released by uct_iface_set_am_handler function.

Return values

<table>
<thead>
<tr>
<th>UCS_OK</th>
<th>descriptor was consumed, and can be released by the caller.</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCS_INPROGRESS</td>
<td>descriptor is owned by the callee, and would be released later. Supported only if flags contain UCT_CB_PARAM_FLAG_DESC value. Otherwise, this is an error.</td>
</tr>
</tbody>
</table>

6.14.2.2 uct_am_tracer_t

typedef void(* uct_am_tracer_t) (void *arg, uct_am_trace_type_t type, uint8_t id, const void *data, size_t length, char *buffer, size_t max)

Writes a string which represents active message contents into 'buffer'.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>arg</th>
<th>User-defined argument.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>type</td>
<td>Message type.</td>
</tr>
<tr>
<td>in</td>
<td>id</td>
<td>Active message id.</td>
</tr>
<tr>
<td>in</td>
<td>data</td>
<td>Points to the received data.</td>
</tr>
<tr>
<td>in</td>
<td>length</td>
<td>Length of data.</td>
</tr>
<tr>
<td>out</td>
<td>buffer</td>
<td>Filled with a debug information string.</td>
</tr>
<tr>
<td>in</td>
<td>max</td>
<td>Maximal length of the string.</td>
</tr>
</tbody>
</table>

6.14.3 Enumeration Type Documentation

6.14.3.1 uct_msg_flags

enum uct_msg_flags

Enumerator

| UCT_SEND_FLAG_SIGNALED | Trigger UCT_EVENT_RECV_SIG event on remote side. Make best effort attempt to avoid triggering UCT_EVENT_RECV event. Ignored if not supported by interface. |
6.14.3.2 uct_am_trace_type

enum uct_am_trace_type

<table>
<thead>
<tr>
<th>Enumerator</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCT_AM TRACE_TYPE_SEND</td>
</tr>
<tr>
<td>UCT_AM TRACE_TYPE_RECV</td>
</tr>
<tr>
<td>UCT_AM TRACE_TYPE_SEND_DROP</td>
</tr>
<tr>
<td>UCT_AM TRACE_TYPE_RECV_DROP</td>
</tr>
<tr>
<td>UCT_AM TRACE_TYPE_TYPE_LAST</td>
</tr>
</tbody>
</table>

6.14.4 Function Documentation

6.14.4.1 uct_iface_set_am_handler()

```c
ucs_status_t uct_iface_set_am_handler (  
    uct_iface_h iface,  
    uint8_t id,  
    uct_am_callback_t cb,  
    void * arg,  
    uint32_t flags)
```

Only one handler can be set of each active message ID, and setting a handler replaces the previous value. If cb == NULL, the current handler is removed.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in <code>iface</code></td>
<td>Interface to set the active message handler for.</td>
</tr>
<tr>
<td>in <code>id</code></td>
<td>Active message id. Must be 0..UCT_AM_ID_MAX-1.</td>
</tr>
<tr>
<td>in <code>cb</code></td>
<td>Active message callback. NULL to clear.</td>
</tr>
<tr>
<td>in <code>arg</code></td>
<td>Active message argument.</td>
</tr>
<tr>
<td>in <code>flags</code></td>
<td>Required callback flags</td>
</tr>
</tbody>
</table>

Returns

- error code if the interface does not support active messages or requested callback flags

Examples

- uct_hello_world.c.

6.14.4.2 uct_iface_set_am_tracer()

```c
ucs_status_t uct_iface_set_am_tracer (  
    uct_iface_h iface,
```
Sets a function which dumps active message debug information to a buffer, which is printed every time an active message is sent or received, when data tracing is on. Without the tracer, only transport-level information is printed.

Parameters

| in  | iface | Interface to set the active message tracer for. |
| in  | tracer | Active message tracer. NULL to clear. |
| in  | arg   | Tracer custom argument. |

6.14.4.3 uct_iface_release_desc()

void uct_iface_release_desc ( void * desc )

Release active message descriptor desc, which was passed to the active message callback, and owned by the callee.

Parameters

| in  | desc | Descriptor to release. |

Examples

uct_hello_world.c.

6.14.4.4 uct_ep_am_short()

ucs_status_t uct_ep_am_short ( uct_ep_h ep, uint8_t id, uint64_t header, const void * payload, unsigned length )

Examples

uct_hello_world.c.

6.14.4.5 uct_ep_am_bcopy()

ssize_t uct_ep_am_bcopy ( uct_ep_h ep, uint8_t id, uct_pack_callback_t pack_cb, void * arg, unsigned flags )
Examples

uct_hello_world.c.

6.14.4.6 uct_ep_am_zcopy()

```c
ucs_status_t uct_ep_am_zcopy (
    uct_ep_h ep,
    uint8_t id,
    const void * header,
    unsigned header_length,
    const uct_iov_t * iov,
    size_t iovcnt,
    unsigned flags,
    uct_completion_t * comp )
```

The input data in `iov` array of `uct_iov_t` structures sent to remote side ("gather output"). Buffers in `iov` are processed in array order. This means that the function complete `iov[0]` before proceeding to `iov[1]`, and so on.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ep</code></td>
<td>Destination endpoint handle.</td>
</tr>
<tr>
<td><code>id</code></td>
<td>Active message id. Must be in range 0..UCT_AM_ID_MAX-1.</td>
</tr>
<tr>
<td><code>header</code></td>
<td>Active message header.</td>
</tr>
<tr>
<td><code>header_length</code></td>
<td>Active message header length in bytes.</td>
</tr>
<tr>
<td><code>iov</code></td>
<td>Points to an array of <code>uct_iov_t</code> structures. The <code>iov</code> pointer must be a valid address of an array of <code>uct_iov_t</code> structures. A particular structure pointer must be a valid address. A NULL terminated array is not required.</td>
</tr>
<tr>
<td><code>iovcnt</code></td>
<td>Size of the <code>iov</code> data <code>uct_iov_t</code> structures array. If <code>iovcnt</code> is zero, the data is considered empty. <code>iovcnt</code> is limited by <code>uct_iface_attr::cap::am::max_iov</code>.</td>
</tr>
<tr>
<td><code>flags</code></td>
<td>Active message flags, see <code>uct_msg_flags</code>.</td>
</tr>
<tr>
<td><code>comp</code></td>
<td>Completion handle as defined by <code>uct_completion_t</code>.</td>
</tr>
</tbody>
</table>

Returns

- **UCS_OK** Operation completed successfully.
- **UCS_INPROGRESS** Some communication operations are still in progress. If non-NULL `comp` is provided, it will be updated upon completion of these operations.
- **UCS_ERR_NO_RESOURCE** Could not start the operation due to lack of send resources.

Note

If the operation returns **UCS_INPROGRESS**, the memory buffers pointed to by `iov` array must not be modified until the operation is completed by `comp`. `header` can be released or changed.

Examples

uct_hello_world.c.
6.15 UCT Remote memory access operations

Functions

- `ucs_status_t uct_ep_put_short (uct_ep_h ep, const void * buffer, unsigned length, uint64_t remote_addr, uct_rkey_t rkey)`
- `ssize_t uct_ep_put_bcopy (uct_ep_h ep, uct_pack_callback_t pack_cb, void * arg, uint64_t remote_addr, uct_rkey_t rkey)`
- `ucs_status_t uct_ep_put_zcopy (uct_ep_h ep, const uct_iov_t * iov, size_t iovcnt, uint64_t remote_addr, uct_rkey_t rkey, uct_completion_t * comp)`

  Write data to remote memory while avoiding local memory copy.

- `ucs_status_t uct_ep_get_short (uct_ep_h ep, void * buffer, unsigned length, uint64_t remote_addr, uct_rkey_t rkey)`
- `ucs_status_t uct_ep_get_bcopy (uct_ep_h ep, uct_unpack_callback_t unpack_cb, void * arg, size_t length, uint64_t remote_addr, uct_rkey_t rkey, uct_completion_t * comp)`
- `ucs_status_t uct_ep_get_zcopy (uct_ep_h ep, const uct_iov_t * iov, size_t iovcnt, uint64_t remote_addr, uct_rkey_t rkey, uct_completion_t * comp)`

  Read data from remote memory while avoiding local memory copy.

6.15.1 Detailed Description

Defines remote memory access operations.

6.15.2 Function Documentation

6.15.2.1 uct_ep_put_short()

```c
ucs_status_t uct_ep_put_short ( 
    uct_ep_h ep, 
    const void * buffer, 
    unsigned length, 
    uint64_t remote_addr, 
    uct_rkey_t rkey )
```

6.15.2.2 uct_ep_put_bcopy()

```c
ssize_t uct_ep_put_bcopy ( 
    uct_ep_h ep, 
    uct_pack_callback_t pack_cb, 
    void * arg, 
    uint64_t remote_addr, 
    uct_rkey_t rkey )
```

6.15.2.3 uct_ep_put_zcopy()

```c
ucs_status_t uct_ep_put_zcopy ( 
    uct_ep_h ep, 
    const uct_iov_t * iov,
```
size_t iovcnt,
uint64_t remote_addr,
uct_rkey_t rkey,
uct_completion_t * comp )

The input data in iov array of uct_iov_t structures sent to remote address ("gather output"). Buffers in iov are processed in array order. This means that the function complete iov[0] before proceeding to iov[1], and so on.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ep</td>
<td>Destination endpoint handle.</td>
</tr>
<tr>
<td>iov</td>
<td>Points to an array of uct_iov_t structures. The iov pointer must be a valid address of an array of uct_iov_t structures. A particular structure pointer must be a valid address. A NULL terminated array is not required.</td>
</tr>
<tr>
<td>iovcnt</td>
<td>Size of the iov data uct_iov_t structures array. If iovcnt is zero, the data is considered empty. iovcnt is limited by uct_iface_attr::cap::put::max_iov.</td>
</tr>
<tr>
<td>remote_addr</td>
<td>Remote address to place the iov data.</td>
</tr>
<tr>
<td>rkey</td>
<td>Remote key descriptor provided by uct_rkey_unpack</td>
</tr>
<tr>
<td>comp</td>
<td>Completion handle as defined by uct_completion_t.</td>
</tr>
</tbody>
</table>

Returns

UCS_INPROGRESS Some communication operations are still in progress. If non-NULL comp is provided, it will be updated upon completion of these operations.

6.15.2.4 uct_ep_get_short()

```c
ucs_status_t uct_ep_get_short ( 
    uct_ep_h ep, 
    void * buffer, 
    unsigned length, 
    uint64_t remote_addr, 
    uct_rkey_t rkey )
```

6.15.2.5 uct_ep_get_bcopy()

```c
ucs_status_t uct_ep_get_bcopy ( 
    uct_ep_h ep, 
    uct_unpack_callback_t unpack_cb, 
    void * arg, 
    size_t length, 
    uint64_t remote_addr, 
    uct_rkey_t rkey, 
    uct_completion_t * comp )
```

6.15.2.6 uct_ep_get_zcopy()

```c
ucs_status_t uct_ep_get_zcopy ( 
    uct_ep_h ep, 
    const uct_iov_t * iov,
```

© 2021 Unified Communication X (UCX). All rights reserved.
size_t iovcnt,
uint64_t remote_addr,
uct_rkey_t rkey,
uct_completion_t * comp)

The output data in `iov` array of `uct_iov_t` structures received from remote address ("scatter input"). Buffers in `iov` are processed in array order. This means that the function complete `iov[0]` before proceeding to `iov[1]`, and so on.

**Parameters**

<table>
<thead>
<tr>
<th>in</th>
<th>ep</th>
<th>Destination endpoint handle.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>iov</td>
<td>Points to an array of <code>uct_iov_t</code> structures. The <code>iov</code> pointer must be a valid address of an array of <code>uct_iov_t</code> structures. A particular structure pointer must be a valid address. A NULL terminated array is not required.</td>
</tr>
<tr>
<td>in</td>
<td>iovcnt</td>
<td>Size of the <code>iov</code> <code>uct_iov_t</code> structures array. If <code>iovcnt</code> is zero, the data is considered empty. <code>iovcnt</code> is limited by <code>uct_iface_attr::cap::get::max_iov</code>.</td>
</tr>
<tr>
<td>in</td>
<td>remote_addr</td>
<td>Remote address of the data placed to the <code>iov</code>.</td>
</tr>
<tr>
<td>in</td>
<td>rkey</td>
<td>Remote key descriptor provided by <code>uct_rkey_unpack</code></td>
</tr>
<tr>
<td>in</td>
<td>comp</td>
<td>Completion handle as defined by <code>uct_completion_t</code>.</td>
</tr>
</tbody>
</table>

**Returns**

`UCS_INPROGRESS` Some communication operations are still in progress. If non-NULL `comp` is provided, it will be updated upon completion of these operations.
6.16 UCT Atomic operations

Functions

- `ucs_status_t uct_ep_atomic_cswap64 (uct_ep_h ep, uint64_t compare, uint64_t swap, uint64_t remote_addr, uct_rkey_t rkey, uint64_t *result, uct_completion_t *comp)`
- `ucs_status_t uct_ep_atomic_cswap32 (uct_ep_h ep, uint32_t compare, uint32_t swap, uint64_t remote_addr, uct_rkey_t rkey, uint32_t *result, uct_completion_t *comp)`
- `ucs_status_t uct_ep_atomic32_post (uct_ep_h ep, uct_atomic_op_t opcode, uint32_t value, uint64_t remote_addr, uct_rkey_t rkey)`
- `ucs_status_t uct_ep_atomic64_post (uct_ep_h ep, uct_atomic_op_t opcode, uint64_t value, uint64_t remote_addr, uct_rkey_t rkey)`
- `ucs_status_t uct_ep_atomic32_fetch (uct_ep_h ep, uct_atomic_op_t opcode, uint32_t value, uint32_t *result, uint64_t remote_addr, uct_rkey_t rkey, uct_completion_t *comp)`
- `ucs_status_t uct_ep_atomic64_fetch (uct_ep_h ep, uct_atomic_op_t opcode, uint64_t value, uint64_t *result, uint64_t remote_addr, uct_rkey_t rkey, uct_completion_t *comp)`

6.16.1 Detailed Description

Defines atomic operations.

6.16.2 Function Documentation

6.16.2.1 uct_ep_atomic_cswap64()

```c
ucs_status_t uct_ep_atomic_cswap64 {
  uct_ep_h ep,
  uint64_t compare,
  uint64_t swap,
  uint64_t remote_addr,
  uct_rkey_t rkey,
  uint64_t *result,
  uct_completion_t *comp
}
```

6.16.2.2 uct_ep_atomic_cswap32()

```c
ucs_status_t uct_ep_atomic_cswap32 {
  uct_ep_h ep,
  uint32_t compare,
  uint32_t swap,
  uint64_t remote_addr,
  uct_rkey_t rkey,
  uint32_t *result,
  uct_completion_t *comp
}
```

6.16.2.3 uct_ep_atomic32_post()

```c
ucs_status_t uct_ep_atomic32_post {
  uct_ep_h ep,
```

6.16.2.4 uct_ep_atomic64_post()

```c
ucs_status_t uct_ep_atomic64_post {
  uct_ep_h ep,
```

6.16.2.5 uct_ep_atomic32_fetch()

```c
ucs_status_t uct_ep_atomic32_fetch {
  uct_ep_h ep,
  uint32_t compare,
  uint32_t swap,
  uint64_t remote_addr,
  uct_rkey_t rkey,
  uint32_t *result,
  uct_completion_t *comp
}
```

6.16.2.6 uct_ep_atomic64_fetch()

```c
ucs_status_t uct_ep_atomic64_fetch {
  uct_ep_h ep,
  uint64_t compare,
  uint64_t swap,
  uint64_t remote_addr,
  uct_rkey_t rkey,
  uint64_t *result,
  uct_completion_t *comp
}
```
6.16 UCT Atomic operations

uct_atomic_op_t opcode,
uint32_t value,
uint64_t remote_addr,
uct_rkey_t rkey )

6.16.2.4 uct_ep_atomic64_post()

ucs_status_t uct_ep_atomic64_post (  
uct_ep_h ep,
uct_atomic_op_t opcode,
uint64_t value,
uint64_t remote_addr,
uct_rkey_t rkey )

6.16.2.5 uct_ep_atomic32_fetch()

ucs_status_t uct_ep_atomic32_fetch (  
uct_ep_h ep,
uct_atomic_op_t opcode,
uint32_t value,
uint32_t * result,
uint64_t remote_addr,
uct_rkey_t rkey,
uct_completion_t * comp )

6.16.2.6 uct_ep_atomic64_fetch()

ucs_status_t uct_ep_atomic64_fetch (  
uct_ep_h ep,
uct_atomic_op_t opcode,
uint64_t value,
uint64_t * result,
uint64_t remote_addr,
uct_rkey_t rkey,
uct_completion_t * comp )

© 2021 Unified Communication X (UCX). All rights reserved.
6.17 UCT Tag matching operations

Data Structures

- struct uct_tag_context
  
  Posted tag context.

Typedefs

- typedef ucs_status_t(uct_tag_unexp_eager_cb_t) (void *arg, void *data, size_t length, unsigned flags, uct_tag_t stag, uint64_t imm, void **context)
  
  Callback to process unexpected eager tagged message.

- typedef ucs_status_t(uct_tag_unexp_rndv_cb_t) (void *arg, unsigned flags, uint64_t stag, const void *header, unsigned header_length, uint64_t remote_addr, size_t length, const void *rkey_buf)
  
  Callback to process unexpected rendezvous tagged message.

Functions

- ucs_status_t uct_ep_tag_eager_short (uct_ep_h ep, uct_tag_t tag, const void *data, size_t length)
  
  Short eager tagged-send operation.

- ssize_t uct_ep_tag_eager_bcopy (uct_ep_h ep, uct_tag_t tag, uint64_t imm, uct_pack_callback_t pack_cb, void *arg, unsigned flags)
  
  Bcopy eager tagged-send operation.

- ucs_status_t uct_ep_tag_eager_zcopy (uct_ep_h ep, uct_tag_t tag, uint64_t imm, const uct_iov_t *iov, size_t iovcnt, unsigned flags, uct_completion_t *comp)
  
  Zcopy eager tagged-send operation.

- ucs_status_ptr_t uct_ep_tag_rndv_zcopy (uct_ep_h ep, uct_tag_t tag, const void *header, unsigned header_length, const uct_iov_t *iov, size_t iovcnt, unsigned flags, uct_completion_t *comp)
  
  Rendezvous tagged-send operation.

- ucs_status_t uct_ep_tag_rndv_cancel (uct_ep_h ep, void *op)
  
  Cancel outstanding rendezvous operation.

- ucs_status_t uct_ep_tag_rndv_request (uct_ep_h ep, uct_tag_t tag, const void *header, unsigned header_length, unsigned flags)
  
  Send software rendezvous request.

- ucs_status_t uct_iface_tag_recv_zcopy (uct_iface_h iface, uct_tag_t tag, uct_tag_t tag_mask, const uct_iov_t *iov, size_t iovcnt, uct_tag_context_t *ctx)
  
  Post a tag to a transport interface.

- ucs_status_t uct_iface_tag_recv_cancel (uct_iface_h iface, uct_tag_context_t *ctx, int force)
  
  Cancel a posted tag.

6.17.1 Detailed Description

Defines tag matching operations.

6.17.2 Typedef Documentation
6.17.2.1 uct_tag_unexp_eager_cb_t

typedef ucs_status_t(∗uct_tag_unexp_eager_cb_t)(void ∗arg, void ∗data, size_t length, unsigned flags, uct_tag_t stag, uint64_t imm, void ∗∗context)

This callback is invoked when tagged message sent by eager protocol has arrived and no corresponding tag has been posted.

Note

The callback is always invoked from the context (thread, process) that called uct_iface_progress(). It is allowed to call other communication routines from the callback.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arg</td>
<td>User-defined argument</td>
</tr>
<tr>
<td>data</td>
<td>Points to the received unexpected data.</td>
</tr>
<tr>
<td>length</td>
<td>Length of data.</td>
</tr>
<tr>
<td>flags</td>
<td>Mask with uct_cb_param_flags flags. If it contains UCT_CB_PARAM_FLAG_DESC value, this means data is part of a descriptor which must be released later using uct_iface_release_desc by the user if the callback returns UCS_INPROGRESS.</td>
</tr>
<tr>
<td>stag</td>
<td>Tag from sender.</td>
</tr>
<tr>
<td>imm</td>
<td>Immediate data from sender.</td>
</tr>
<tr>
<td>context</td>
<td>Storage for a per-message user-defined context. In this context, the message is defined by the sender side as a single call to uct_ep_tag_eager_short/bcopy/zcopy. On the transport level the message can be fragmented and delivered to the target over multiple fragments. The fragments will preserve the original order of the message. Each fragment will result in invocation of the above callback. The user can use UCT_CB_PARAM_FLAG_FIRST to identify the first fragment, allocate the context object and use the context as a token that is set by the user and passed to subsequent callbacks of the same message. The user is responsible for allocation and release of the context.</td>
</tr>
</tbody>
</table>

Note

No need to allocate the context in the case of a single fragment message (i.e. flags contains UCT_CB_PARAM_FLAG_FIRST, but does not contain UCT_CB_PARAM_FLAG_MORE).

Return values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCS_OK</td>
<td>- data descriptor was consumed, and can be released by the caller.</td>
</tr>
<tr>
<td>UCS_INPROGRESS</td>
<td>- data descriptor is owned by the callee, and will be released later.</td>
</tr>
</tbody>
</table>

6.17.2.2 uct_tag_unexp_rndv_cb_t

typedef ucs_status_t(∗uct_tag_unexp_rndv_cb_t)(void ∗arg, unsigned flags, uint64_t stag, const void ∗header, unsigned header_length, uint64_t remote_addr, size_t length, const void ∗rkey_buf)

This callback is invoked when rendezvous send notification has arrived and no corresponding tag has been posted.
Note

The callback is always invoked from the context (thread, process) that called `uct_iface_progress()`. It is allowed to call other communication routines from the callback.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>arg</th>
<th>User-defined argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>flags</td>
<td>Mask with <code>uct_cb_param_flags</code></td>
</tr>
<tr>
<td>in</td>
<td>stag</td>
<td>Tag from sender.</td>
</tr>
<tr>
<td>in</td>
<td>header</td>
<td>User defined header.</td>
</tr>
<tr>
<td>in</td>
<td>header_length</td>
<td>User defined header length in bytes.</td>
</tr>
<tr>
<td>in</td>
<td>remote_addr</td>
<td>Sender's buffer virtual address.</td>
</tr>
<tr>
<td>in</td>
<td>length</td>
<td>Sender's buffer length.</td>
</tr>
<tr>
<td>in</td>
<td>rkey_buf</td>
<td>Sender's buffer packed remote key. It can be passed to <code>uct_rkey_unpack()</code> to create <code>uct_rkey_t</code>.</td>
</tr>
</tbody>
</table>

Warning

If the user became the owner of the `desc` (by returning `UCS_INPROGRESS`) the descriptor must be released later by `uct_iface_release_desc` by the user.

Return values

<table>
<thead>
<tr>
<th><code>UCS_OK</code></th>
<th>- descriptor was consumed, and can be released by the caller.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>UCS_INPROGRESS</code></td>
<td>- descriptor is owned by the callee, and would be released later.</td>
</tr>
</tbody>
</table>

6.17.3 Function Documentation

6.17.3.1 `uct_ep_tag_eager_short()`

```c
ucs_status_t uct_ep_tag_eager_short (  
    uct_ep_h ep,  
    uct_tag_t tag,  
    const void * data,  
    size_t length )
```

This routine sends a message using short eager protocol. Eager protocol means that the whole data is sent to the peer immediately without any preceding notification. The data is provided as buffer and its length, and must not be larger than the corresponding `max_short` value in `uct_iface_attr`. The immediate value delivered to the receiver is implicitly equal to 0. If it’s required to pass nonzero imm value, `uct_ep_tag_eager_bcopy` should be used.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>ep</th>
<th>Destination endpoint handle.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>tag</td>
<td>Tag to use for the eager message.</td>
</tr>
<tr>
<td>in</td>
<td>data</td>
<td>Data to send.</td>
</tr>
<tr>
<td>in</td>
<td>length</td>
<td>Data length.</td>
</tr>
</tbody>
</table>
6.17 UCT Tag matching operations

Returns

UCS_OK - operation completed successfully.
UCS_ERR_NO_RESOURCE - could not start the operation due to lack of send resources.

6.17.3.2 uct_ep_tag_eager_bcopy()

ssize_t uct_ep_tag_eager_bcopy (uct_ep_h ep, uct_tag_t tag, uint64_t imm, uct_pack_callback_t pack_cb, void * arg, unsigned flags)

This routine sends a message using bcopy eager protocol. Eager protocol means that the whole data is sent to the peer immediately without any preceding notification. Custom data callback is used to copy the data to the network buffers.

Note

The resulted data length must not be larger than the corresponding max_bcopy value in uct_iface_attr.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in ep</td>
<td>Destination endpoint handle.</td>
</tr>
<tr>
<td>in tag</td>
<td>Tag to use for the eager message.</td>
</tr>
<tr>
<td>in imm</td>
<td>Immediate value which will be available to the receiver.</td>
</tr>
<tr>
<td>in pack_cb</td>
<td>User callback to pack the data.</td>
</tr>
<tr>
<td>in arg</td>
<td>Custom argument to pack_cb.</td>
</tr>
<tr>
<td>in flags</td>
<td>Tag message flags, see uct_msg_flags.</td>
</tr>
</tbody>
</table>

Returns

>=0 - The size of the data packed by pack_cb.
otherwise - Error code.

6.17.3.3 uct_ep_tag_eager_zcopy()

ucs_status_t uct_ep_tag_eager_zcopy (uct_ep_h ep, uct_tag_t tag, uint64_t imm, const uct_iov_t * iov, size_t iovcnt, unsigned flags, uct_completion_t * comp)

This routine sends a message using zcopy eager protocol. Eager protocol means that the whole data is sent to the peer immediately without any preceding notification. The input data (which has to be previously registered) in iov array of uct_iov_t structures sent to remote side ("gather output"). Buffers in iov are processed in array order, so the function complete iov[0] before proceeding to iov[1], and so on.
Note

The resulted data length must not be larger than the corresponding max_zcopy value in uct_iface_attr.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in ep</td>
<td>Destination endpoint handle.</td>
</tr>
<tr>
<td>in tag</td>
<td>Tag to use for the eager message.</td>
</tr>
<tr>
<td>in imm</td>
<td>Immediate value which will be available to the receiver.</td>
</tr>
<tr>
<td>in iov</td>
<td>Points to an array of uct_iov_t structures. A particular structure pointer must be valid address. A NULL terminated array is not required.</td>
</tr>
<tr>
<td>in iovcnt</td>
<td>Size of the iov array. If iovcnt is zero, the data is considered empty. Note that iovcnt is limited by the corresponding max_iov value in uct_iface_attr.</td>
</tr>
<tr>
<td>in flags</td>
<td>Tag message flags, see uct_msg_flags.</td>
</tr>
<tr>
<td>in comp</td>
<td>Completion callback which will be called when the data is reliably received by the peer, and the buffer can be reused or invalidated.</td>
</tr>
</tbody>
</table>

Returns

UCS_OK - operation completed successfully.
UCS_ERR_NO_RESOURCE - could not start the operation due to lack of send resources.
UCS_INPROGRESS - operation started, and comp will be used to notify when it's completed.

6.17.3.4 uct_ep_tag_rndv_zcopy()

This routine sends a message using rendezvous protocol. Rendezvous protocol means that only a small notification is sent at first, and the data itself is transferred later (when there is a match) to avoid extra memory copy.

Note

The header will be available to the receiver in case of unexpected rendezvous operation only, i.e. the peer has not posted tag for this message yet (by means of uct_iface_tag_recv_zcopy), when it is arrived.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in ep</td>
<td>Destination endpoint handle.</td>
</tr>
<tr>
<td>in tag</td>
<td>Tag to use for the eager message.</td>
</tr>
<tr>
<td>in header</td>
<td>User defined header.</td>
</tr>
<tr>
<td>in header_length</td>
<td>User defined header length in bytes. Note that it is limited by the corresponding max_hdr value in uct_iface_attr.</td>
</tr>
<tr>
<td>in iov</td>
<td>Points to an array of uct_iov_t structures. A particular structure pointer must be valid address. A NULL terminated array is not required.</td>
</tr>
<tr>
<td>in iovcnt</td>
<td>Size of the iov array. If iovcnt is zero, the data is considered empty. Note that iovcnt is limited by the corresponding max_iov value in uct_iface_attr.</td>
</tr>
</tbody>
</table>
Parameters

| in | flags | Tag message flags, see uct_msg_flags. |
| in | comp | Completion callback which will be called when the data is reliably received by the peer, and the buffer can be reused or invalidated. |

Returns

\( \geq 0 \) - The operation is in progress and the return value is a handle which can be used to cancel the outstanding rendezvous operation.
otherwise - Error code.

6.17.3.5 uct_ep_tag_rndv_cancel()

```c
ucs_status_t uct_ep_tag_rndv_cancel ( 
    uct_ep_h ep, 
    void * op )
```

This routine signals the underlying transport disregard the outstanding operation without calling completion callback provided in uct_ep_tag_rndv_zcopy.

Note

The operation handle should be valid at the time the routine is invoked. I.e. it should be a handle of the real operation which is not completed yet.

Parameters

| in | ep | Destination endpoint handle. |
| in | op | Rendezvous operation handle, as returned from uct_ep_tag_rndv_zcopy. |

Returns

UCS_OK - The operation has been canceled.

6.17.3.6 uct_ep_tag_rndv_request()

```c
ucs_status_t uct_ep_tag_rndv_request ( 
    uct_ep_h ep, 
    uct_tag_t tag, 
    const void * header, 
    unsigned header_length, 
    unsigned flags )
```

This routine sends a rendezvous request only, which indicates that the data transfer should be completed in software.

Parameters

| in | ep | Destination endpoint handle. |
| in | tag | Tag to use for matching. |
Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>header</th>
<th>User defined header</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>header_length</td>
<td>User defined header length in bytes. Note that it is limited by the corresponding max_hdr value in uct_iface_attr.</td>
</tr>
<tr>
<td>in</td>
<td>flags</td>
<td>Tag message flags, see uct_msg_flags.</td>
</tr>
</tbody>
</table>

Returns

UCS_OK - operation completed successfully.
UCS_ERR_NO_RESOURCE - could not start the operation due to lack of send resources.

6.17.3.7 uct_iface_tag_recv_zcopy()

```c
ucs_status_t uct_iface_tag_recv_zcopy (uct_iface_h iface, uct_tag_t tag, uct_tag_t tag_mask, const uct_iov_t *iov, size_t iovcnt, uct_tag_context_t *ctx)
```

This routine posts a tag to be matched on a transport interface. When a message with the corresponding tag arrives it is stored in the user buffer (described by iov and iovcnt) directly. The operation completion is reported using callbacks on the ctx structure.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>iface</th>
<th>Interface to post the tag on.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>tag</td>
<td>Tag to expect.</td>
</tr>
<tr>
<td>in</td>
<td>tag_mask</td>
<td>Mask which specifies what bits of the tag to compare.</td>
</tr>
<tr>
<td>in</td>
<td>iov</td>
<td>Points to an array of uct_iov_t structures. The iov pointer must be a valid address of an array of uct_iov_t structures. A particular structure pointer must be a valid address. A NULL terminated array is not required.</td>
</tr>
<tr>
<td>in</td>
<td>iovcnt</td>
<td>Size of the iov data uct_iov_t structures array. If iovcnt is zero, the data is considered empty. iovcnt is limited by uct_iface_attr::cap::tag::max_iov.</td>
</tr>
<tr>
<td>in, out</td>
<td>ctx</td>
<td>Context associated with this particular tag, “priv” field in this structure is used to track the state internally.</td>
</tr>
</tbody>
</table>

Returns

UCS_OK - The tag is posted to the transport.
UCS_ERR_NO_RESOURCE - Could not start the operation due to lack of resources.
UCS_ERR_EXCEEDS_LIMIT - No more room for tags in the transport.

6.17.3.8 uct_iface_tag_recv_cancel()

```c
ucs_status_t uct_iface_tag_recv_cancel (uct_iface_h iface,
```
This routine cancels a tag, which was previously posted by `uct_iface_tag_recv_zcopy`. The tag would be either matched or canceled, in a bounded time, regardless of the peer actions. The original completion callback of the tag would be called with the status if `force` is not set.

### Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>iface</th>
<th>Interface to cancel the tag on.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>ctx</td>
<td>Tag context which was used for posting the tag. If force is 0, <code>ctx-&gt;completed_cb</code> will be called with either UCS_OK which means the tag was matched and data received despite the cancel request, or UCS_ERR_CANCELED which means the tag was successfully canceled before it was matched.</td>
</tr>
<tr>
<td>in</td>
<td>force</td>
<td>Whether to report completions to <code>ctx-&gt;completed_cb</code>. If nonzero, the cancel is assumed to be successful, and the callback is not called.</td>
</tr>
</tbody>
</table>

### Returns

UCS_OK - The tag is canceled in the transport.
6.18 UCT client-server operations

Data Structures

- `struct uct_cm_attr`
  Connection manager attributes, capabilities and limitations. More...
- `struct uct_listener_attr`
  UCT listener attributes, capabilities and limitations. More...
- `struct uct_listener_params`
  Parameters for creating a listener object `uct_listener_h` by `uct_listener_create`. More...
- `struct uct_cm_ep_priv_data_pack_args`
  Arguments to the client-server private data pack callback. More...
- `struct uct_cm_remote_data`
  Data received from the remote peer. More...
- `struct uct_listener_conn_request_args`
  Arguments to the listener's connection request callback. More...
- `struct uct_cm_ep_client_connect_args`
  Arguments to the client's connect callback. More...
- `struct uct_cm_ep_server_conn_notify_args`
  Arguments to the server's notify callback. More...

Typedefs

- `typedef struct uct_cm_ep_priv_data_pack_args uct_cm_ep_priv_data_pack_args_t`
  Arguments to the client-server private data pack callback.
- `typedef struct uct_cm_remote_data uct_cm_remote_data_t`
  Data received from the remote peer.
- `typedef struct uct_cm_listener_conn_request_args uct_cm_listener_conn_request_args_t`
  Arguments to the listener's connection request callback.
- `typedef struct uct_cm_ep_client_connect_args uct_cm_ep_client_connect_args_t`
  Arguments to the client's connect callback.
- `typedef struct uct_cm_ep_server_conn_notify_args uct_cm_ep_server_conn_notify_args_t`
  Arguments to the server's notify callback.
- `typedef void(uct_sockaddr_conn_request_callback_t) (uct_iface_h iface, void *arg, uct_conn_request_h conn_request, const void *conn_priv_data, size_t length)`
  Callback to process an incoming connection request on the server side.
- `typedef void(uct_cm_listener_conn_request_callback_t) (uct_listener_h listener, void *arg, const uct_cm_listener_conn_request_args_t *conn_req_args)`
  Callback to process an incoming connection request on the server side listener in a connection manager.
- `typedef void(uct_cm_ep_client_connect_callback_t) (uct_ep_h ep, void *arg, const uct_cm_ep_client_connect_args_t *connect_args)`
  Callback to process an incoming connection establishment acknowledgment on the server side listener, from the client, which indicates that the client side is connected. The callback also notifies the server side of a local error on a not-yet-connected endpoint.
- `typedef void(uct_cm_ep_disconnect_cb_t) (uct_ep_h ep, void *arg)`
  Callback to handle the disconnection of the remote peer.
- `typedef ssize_t(uct_cm_ep_priv_data_pack_callback_t) (void *arg, const uct_cm_ep_priv_data_pack_args_t *pack_args, void *priv_data)`
  Callback to fill the user's private data in a client-server flow.
Enumerations

- enum uct_cm_attr_field { UCT_CM_ATTR_FIELD_MAX_CONN_PRIV = UCS_BIT(0) }
  UCT connection manager attributes field mask.
- enum uct_listener_attr_field { UCT_LISTENER_ATTR_FIELD_SOCKADDR = UCS_BIT(0) }
  UCT listener attributes field mask.
- enum uct_listener_params_field { UCT_LISTENER_PARAM_FIELD_BACKLOG = UCS_BIT(0), UCT_LISTENER_PARAM_FIELD_USER_DATA = UCS_BIT(2) }
  UCT listener created by uct_listener_create parameters field mask.
- enum uct_cm_ep_priv_data_pack_args_field { UCT_CM_EP_PRIV_DATA_PACK_ARGS_FIELD_DEVICE_NAME = UCS_BIT(0) }
  Client-Server private data pack callback arguments field mask.
- enum uct_cm_remote_data_field { UCT_CM_REMOTE_DATA_FIELD_DEV_ADDR = UCS_BIT(0), UCT_CM_REMOTE_DATA_FIELD_CONN_PRIV_DATA = UCS_BIT(2), UCT_CM_REMOTE_DATA_FIELD_CONN_PRIV_DATA_LENGTH = UCS_BIT(3) }
  Remote data attributes field mask.
- enum uct_cm_listener_conn_request_args_field { UCT_CM_LISTENER_CONN_REQUEST_ARGS_FIELD_DEV_NAME = UCS_BIT(0), UCT_CM_LISTENER_CONN_REQUEST_ARGS_FIELD_CONN_REQUEST = UCS_BIT(1), UCT_CM_LISTENER_CONN_REQUEST_ARGS_FIELD_REMOTE_DATA = UCS_BIT(2), UCT_CM_LISTENER_CONN_REQUEST_ARGS_FIELD_CLIENT_ADDR = UCS_BIT(3) }
  Listener's connection request callback arguments field mask.
- enum uct_cm_ep_client_connect_args_field { UCT_CM_EP_CLIENT_CONNECT_ARGS_FIELD_REMOTE_DATA = UCS_BIT(0), UCT_CM_EP_CLIENT_CONNECT_ARGS_FIELD_STATUS = UCS_BIT(1) }
  Field mask flags for client-side connection established callback.
- enum uct_cm_ep_server_conn_notify_args_field { UCT_CM_EP_SERVER_CONN_NOTIFY_ARGS_FIELD_STATUS = UCS_BIT(0) }
  Field mask flags for server-side connection established notification callback.

Functions

- ucs_status_t uct_iface_accept (uct_iface_h iface, uct_conn_request_h conn_request)
  Accept connection request.
- ucs_status_t uct_iface_reject (uct_iface_h iface, uct_conn_request_h conn_request)
  Reject connection request. Will invoke an error handler uct_error_handler_t on the remote transport interface, if set.
- ucs_status_t uct_ep_disconnect (uct_ep_h ep, unsigned flags)
  Initiate a disconnection of an endpoint connected to a sockaddr by a connection manager uct_cm_h.
- ucs_status_t uct_cm_open (uct_component_h component, uct_worker_h worker, const uct_cm_config_t *config, uct_cm_h *cm_p)
  Open a connection manager.
- void uct_cm_close (uct_cm_h cm)
  Close a connection manager.
- ucs_status_t uct_cm_query (uct_cm_h cm, uct_cm_attr_t *cm_attr)
  Get connection manager attributes.
- ucs_status_t uct_cm_config_read (uct_component_h component, const char *env_prefix, const char *filename, uct_cm_config_t **config_p)
  Read the configuration for a connection manager.
- ucs_status_t uct_cm_client_ep_conn_notify (uct_ep_h ep)
  Notify the server about client-side connection establishment.
- ucs_status_t uct_listener_create (uct_cm_h cm, const struct sockaddr *saddr, socklen_t socklen, const uct_listener_params_t *params, uct_listener_h *listener_p)
  Create a new transport listener object.
- void uct_listener_destroy (uct_listener_h listener)
Module Documentation

Destroy a transport listener.

- `ucs_status_t uct_listener_reject (uct_listener_h listener, uct_conn_request_h conn_request)`
  Reject a connection request.

- `ucs_status_t uct_listener_query (uct_listener_h listener, uct_listener_attr_t *listener_attr)`
  Get attributes specific to a particular listener.

### 6.18.1 Detailed Description

Defines client-server operations. The client-server API allows the connection establishment between an active side - a client, and its peer - the passive side - a server. The connection can be established through a UCT transport that supports listening and connecting via IP address and port (listening can also be on INADDR_ANY).

The following is a general overview of the operations on the server side:

Connecting: `uct_cm_open` Open a connection manager. `uct_listener_create` Create a listener on the CM and start listening on a given IP:port / INADDR_ANY. `uct_cm_listener_conn_request_callback_t` This callback is invoked by the UCT transport to handle an incoming connection request from a client. Accept or reject the client's connection request. `uct_ep_create` Connect to the client by creating an endpoint if the request is accepted. The server creates a new endpoint for every connection request that it accepts. `uct_cm_ep_priv_data_pack_callback_t` This callback is invoked by the UCT transport to fill auxiliary data in the connection acknowledgement or reject notification back to the client. Send the client a connection acknowledgement or reject notification. Wait for an acknowledgment from the client, indicating that it is connected. `uct_cm_ep_server_conn_notify_callback_t` This callback is invoked by the UCT transport to handle the connection notification from the client.

Disconnecting: `uct_ep_disconnect` Disconnect the server's endpoint from the client. Can be called when initiating a disconnect or when receiving a disconnect notification from the remote side. `uct_ep_disconnect_cb_t` This callback is invoked by the UCT transport when the client side calls `uct_ep_disconnect` as well. `uct_ep_destroy` Destroy the endpoint connected to the remote peer. If this function is called before the endpoint was disconnected, the `uct_ep_disconnect_cb_t` will not be invoked.

Destroying the server's resources: `uct_listener_destroy` Destroy the listener object. `uct_cm_close` Close the connection manager.

The following is a general overview of the operations on the client side:

Connecting: `uct_cm_open` Open a connection manager. `uct_ep_create` Create an endpoint for establishing a connection to the server. `uct_cm_ep_priv_data_pack_callback_t` This callback is invoked by the UCT transport to fill the user's private data in the connection request to be sent to the server. This connection request should be created by the transport. Send the connection request to the server. Wait for an acknowledgment from the server, indicating that it is connected. `uct_cm_ep_client_connect_callback_t` This callback is invoked by the UCT transport to handle a connection response from the server. After invoking this callback, the UCT transport will finalize the client's connection to the server. `uct_cm_client_ep_conn_notify` After the client's connection establishment is completed, the client should call this function in which it sends a notification message to the server stating that it (the client) is connected. The notification message that is sent depends on the transport's implementation.

Disconnecting: `uct_ep_disconnect` Disconnect the client's endpoint from the server. Can be called when initiating a disconnect or when receiving a disconnect notification from the remote side. `uct_ep_disconnect_cb_t` This callback is invoked by the UCT transport when the server side calls `uct_ep_disconnect` as well. `uct_ep_destroy` Destroy the endpoint connected to the remote peer.

Destroying the client's resources: `uct_cm_close` Close the connection manager.

### 6.18.2 Data Structure Documentation

#### 6.18.2.1 struct uct_cm_attr

**Data Fields**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>field_mask</td>
<td>Mask of valid fields in this structure, using bits from <code>uct_cm_attr_field</code>. Fields not specified by this mask will be ignored.</td>
</tr>
</tbody>
</table>
6.18 UCT client-server operations

Data Fields

| size_t | max_conn_priv | Max size of the connection manager’s private data used for connection establishment with sockaddr. |

6.18.2.2 struct uct_listener_attr

Data Fields

| uint64_t | field_mask | Mask of valid fields in this structure, using bits from uct_listener_attr_field. Fields not specified by this mask will be ignored. |
| struct sockaddr_storage | sockaddr | Sockaddr on which this listener is listening. |

6.18.2.3 struct uct_listener_params

Data Fields

| uint64_t | field_mask | Mask of valid fields in this structure, using bits from uct_listener_params_field. Fields not specified by this mask will be ignored. |
| int | backlog | Backlog of incoming connection requests. If specified, must be a positive value. If not specified, each CM component will use its maximal allowed value, based on the system’s setting. |
| uct_cm_listener_conn_request_callback_t | conn_request_cb | Callback function for handling incoming connection requests. |
| void * | user_data | User data associated with the listener. |

6.18.2.4 struct uct_cm_ep_priv_data_pack_args

Used with the client-server API on a connection manager.

Data Fields

| uint64_t | field_mask | Mask of valid fields in this structure, using bits from uct_cm_ep_priv_data_pack_args_field. Fields not specified by this mask should not be accessed by the callback. |
| char [UCT_DEVICE_NAME_MAX] | dev_name | Device name. This routine may fill the user’s private data according to the given device name. The device name that is passed to this routine, corresponds to uct_tl_resource_desc_t::dev_name as returned from uct_md_query_tl_resources. |

6.18.2.5 struct uct_cm_remote_data

The remote peer’s device address, the data received from it and their lengths. Used with the client-server API on a connection manager.
Data Fields

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint64_t</td>
<td>field_mask</td>
<td>Mask of valid fields in this structure, using bits from <code>uct_cm_remote_data_field</code>. Fields not specified by this mask will be ignored.</td>
</tr>
<tr>
<td>const uct_device_addr_t*</td>
<td>dev_addr</td>
<td>Device address of the remote peer.</td>
</tr>
<tr>
<td>size_t</td>
<td>dev_addr_length</td>
<td>Length of the remote device address.</td>
</tr>
<tr>
<td>const void*</td>
<td>conn_priv_data</td>
<td>Pointer to the received data. This is the private data that was passed to <code>uct_ep_params_t::sockaddr_pack_cb</code>.</td>
</tr>
<tr>
<td>size_t</td>
<td>conn_priv_data_length</td>
<td>Length of the received data from the peer.</td>
</tr>
</tbody>
</table>

6.18.2.6 struct uct_cm_listener_conn_request_args

The local device name, connection request handle and the data the client sent. Used with the client-server API on a connection manager.

Data Fields

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint64_t</td>
<td>field_mask</td>
<td>Mask of valid fields in this structure, using bits from <code>uct_cm_listener_conn_request_args_field</code>. Fields not specified by this mask should not be accessed by the callback.</td>
</tr>
<tr>
<td>char</td>
<td>dev_name[UCT_DEVICE_NAME_MAX]</td>
<td>Local device name which handles the incoming connection request.</td>
</tr>
<tr>
<td>uct_conn_request_h</td>
<td>conn_request</td>
<td>Connection request handle. Can be passed to this callback from the transport and will be used by it to accept or reject the connection request from the client.</td>
</tr>
<tr>
<td>const uct_cm_remote_data_t*</td>
<td>remote_data</td>
<td>Remote data from the client.</td>
</tr>
<tr>
<td>ucs_sock_addr_t</td>
<td>client_address</td>
<td>Client's address.</td>
</tr>
</tbody>
</table>

6.18.2.7 struct uct_cm_ep_client_connect_args

Used with the client-server API on a connection manager.

Data Fields

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint64_t</td>
<td>field_mask</td>
<td>Mask of valid fields in this structure, using bits from <code>uct_cm_ep_client_connect_args_field</code>. Fields not specified by this mask should not be accessed by the callback.</td>
</tr>
<tr>
<td>const uct_cm_remote_data_t*</td>
<td>remote_data</td>
<td>Remote data from the server.</td>
</tr>
<tr>
<td>ucs_status_t</td>
<td>status</td>
<td>Indicates the connection establishment response from the remote server: UCS_OK - the remote server accepted the connection request. UCS_ERR_REJECTED - the remote server rejected the connection request. UCS_ERR_CONNECTION_RESET - the server's connection was reset during the connection establishment to the client. Otherwise - indicates an internal connection establishment error on the local (client) side.</td>
</tr>
</tbody>
</table>
6.18 UCT client-server operations

6.18.2.8 struct uct_cm_ep_server_conn_notify_args

Used with the client-server API on a connection manager.

Data Fields

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint64_t</td>
<td>field_mask</td>
<td>Mask of valid fields in this structure, using bits from uct_cm_ep_server_conn_notify_args_field. Fields not specified by this mask should not be accessed by the callback.</td>
</tr>
<tr>
<td>ucs_status_t</td>
<td>status</td>
<td>Indicates the client's ucs_status_t status: UCS_OK - the client completed its connection establishment and called uct_cm_client_ep_conn_notify. UCS_ERR_CONNECTION_RESET - the client's connection was reset during the connection establishment to the server. Otherwise - indicates an internal connection establishment error on the local (server) side.</td>
</tr>
</tbody>
</table>

6.18.3 Typedef Documentation

6.18.3.1 uct_cm_ep_priv_data_pack_args_t

typedef struct uct_cm_ep_priv_data_pack_args uct_cm_ep_priv_data_pack_args_t

Used with the client-server API on a connection manager.

6.18.3.2 uct_cm_remote_data_t

typedef struct uct_cm_remote_data uct_cm_remote_data_t

The remote peer's device address, the data received from it and their lengths. Used with the client-server API on a connection manager.

6.18.3.3 uct_cm_listener_conn_request_args_t

typedef struct uct_cm_listener_conn_request_args uct_cm_listener_conn_request_args_t

The local device name, connection request handle and the data the client sent. Used with the client-server API on a connection manager.

6.18.3.4 uct_cm_ep_client_connect_args_t

typedef struct uct_cm_ep_client_connect_args uct_cm_ep_client_connect_args_t

Used with the client-server API on a connection manager.

6.18.3.5 uct_cm_ep_server_conn_notify_args_t

typedef struct uct_cm_ep_server_conn_notify_args uct_cm_ep_server_conn_notify_args_t

Used with the client-server API on a connection manager.
6.18.3.6 uct_sockaddr_conn_request_callback_t

```c
typedef void(* uct_sockaddr_conn_request_callback_t) (uct_iface_h iface, void *arg, uct_conn_request_h conn_request, const void *conn_priv_data, size_t length)
```

This callback routine will be invoked on the server side upon receiving an incoming connection request. It should be set by the server side while initializing an interface. Incoming data is placed inside the conn_priv_data buffer. This callback has to be thread safe. Other than communication progress routines, it is allowed to call other UCT communication routines from this callback.

**Parameters**

<table>
<thead>
<tr>
<th>in</th>
<th>iface</th>
<th>Transport interface.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>arg</td>
<td>User defined argument for this callback.</td>
</tr>
<tr>
<td>in</td>
<td>conn_request</td>
<td>Transport level connection request. The user should accept or reject the request by calling uct_iface_accept or uct_iface_reject routines respectively. conn_request should not be used outside the scope of this callback.</td>
</tr>
<tr>
<td>in</td>
<td>conn_priv_data</td>
<td>Points to the received data. This is the private data that was passed to the uct_ep_params_t::sockaddr_pack_cb on the client side.</td>
</tr>
<tr>
<td>in</td>
<td>length</td>
<td>Length of the received data.</td>
</tr>
</tbody>
</table>

6.18.3.7 uct_cm_listener_conn_request_callback_t

```c
typedef void(* uct_cm_listener_conn_request_callback_t) (uct_listener_h listener, void *arg, const uct_cm_listener_conn_request_args_t *conn_req_args)
```

This callback routine will be invoked on the server side upon receiving an incoming connection request. It should be set by the server side while initializing a listener in a connection manager. This callback has to be thread safe. Other than communication progress routines, it is allowed to call other UCT communication routines from this callback.

**Parameters**

<table>
<thead>
<tr>
<th>in</th>
<th>listener</th>
<th>Transport listener.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>arg</td>
<td>User argument for this callback as defined in uct_listener_params_t::user_data</td>
</tr>
<tr>
<td>in</td>
<td>conn_req_args</td>
<td>Listener’s arguments to handle the connection request from the client.</td>
</tr>
</tbody>
</table>

6.18.3.8 uct_cm_ep_server_conn_notify_callback_t

```c
typedef void(* uct_cm_ep_server_conn_notify_callback_t) (uct_ep_h ep, void *arg, const uct_cm_ep_server_conn_notify_args_t *connect_args)
```

This callback routine will be invoked on the server side upon receiving an incoming connection establishment acknowledgment from the client, which is sent from it once the client is connected to the server. Used to connect the server side to the client or handle an error from it - depending on the status field. This callback will also be invoked in the event of an internal local error with a failed uct_cm_ep_server_conn_notify_args::status if the endpoint was not connected yet. This callback has to be thread safe. Other than communication progress routines, it is permissible to call other UCT communication routines from this callback.

**Parameters**

<table>
<thead>
<tr>
<th>in</th>
<th>ep</th>
<th>Transport endpoint.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>arg</td>
<td>User argument for this callback as defined in uct_ep_params_t::user_data</td>
</tr>
</tbody>
</table>
6.18 UCT client-server operations

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>connect_args</code></td>
<td>Server's connect callback arguments.</td>
</tr>
</tbody>
</table>

6.18.3.9 uct_cm_ep_client_connect_callback_t

typedef void(*uct_cm_ep_client_connect_callback_t) (uct_ep_h ep, void *arg, const uct_cm_ep_client_connect_args_t *connect_args)

This callback routine will be invoked on the client side upon receiving an incoming connection response from the server. Used to connect the client side to the server or handle an error from it - depending on the status field. This callback will also be invoked in the event of an internal local error with a failed `uct_cm_ep_client_connect_args::status` if the endpoint was not connected yet. This callback has to be thread safe. Other than communication progress routines, it is permissible to call other UCT communication routines from this callback.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ep</code></td>
<td>Transport endpoint.</td>
</tr>
<tr>
<td><code>arg</code></td>
<td>User argument for this callback as defined in <code>uct_ep_params_t::user_data</code>.</td>
</tr>
<tr>
<td><code>connect_args</code></td>
<td>Client's connect callback arguments</td>
</tr>
</tbody>
</table>

6.18.3.10 uct_ep_disconnect_cb_t

typedef void(*uct_ep_disconnect_cb_t) (uct_ep_h ep, void *arg)

This callback routine will be invoked on the client and server sides upon a disconnect of the remote peer. It will disconnect the given endpoint from the remote peer. This callback won't be invoked if the endpoint was not connected to the remote peer yet. This callback has to be thread safe. Other than communication progress routines, it is permissible to call other UCT communication routines from this callback.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ep</code></td>
<td>Transport endpoint to disconnect.</td>
</tr>
<tr>
<td><code>arg</code></td>
<td>User argument for this callback as defined in <code>uct_ep_params_t::user_data</code>.</td>
</tr>
</tbody>
</table>

6.18.3.11 uct_cm_ep_priv_data_pack_callback_t

typedef ssize_t(*uct_cm_ep_priv_data_pack_callback_t) (void *arg, const uct_cm_ep_priv_data_pack_args_t *pack_args, void *priv_data)

This callback routine will be invoked on the client side, before sending the transport's connection request to the server, or on the server side before sending a connection response to the client. The callback routine must be set when creating an endpoint. The user's private data should be placed inside the `priv_data` buffer to be sent to the remote side. The maximal allowed length of the private data is indicated by the field `max_connPriv` inside `uct_iface_attr` or inside `uct_cm_attr` when using a connection manager. Communication progress routines should not be called from this callback. It is allowed to call other UCT communication routines from this callback.
Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>arg</th>
<th>User defined argument for this callback.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>pack_args</td>
<td>Handle for the the private data packing.</td>
</tr>
<tr>
<td>out</td>
<td>priv_data</td>
<td>User’s private data to be passed to the remote side.</td>
</tr>
</tbody>
</table>

Returns

Negative value indicates an error according to ucs_status_t. On success, a non-negative value indicates actual number of bytes written to the priv_data buffer.

6.18.4 Enumeration Type Documentation

6.18.4.1 uct_cm_attr_field

defined uct_cm_attr_field

The enumeration allows specifying which fields in uct_cm_attr_t are present, for backward compatibility support.

Enumerator

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Enables</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCT_CM_ATTR_FIELD_MAX_CONN_PRIV</td>
<td>uct_cm_attr::max_conn_priv</td>
</tr>
</tbody>
</table>

6.18.4.2 uct_listener_attr_field

defined uct_listener_attr_field

The enumeration allows specifying which fields in uct_listener_attr_t are present, for backward compatibility support.

Enumerator

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Enables</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCT_LISTENER_ATTR_FIELD_SOCKADDR</td>
<td>uct_listener_attr::sockaddr</td>
</tr>
</tbody>
</table>

6.18.4.3 uct_listener_params_field

defined uct_listener_params_field

The enumeration allows specifying which fields in uct_listener_params_t are present, for backward compatibility support.

Enumerator

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Enables</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCT_LISTENER_PARAM_FIELD_BACKLOG</td>
<td>uct_listener_params::backlog</td>
</tr>
<tr>
<td>UCT_LISTENER_PARAM_FIELD_CONN_REQUEST_CB</td>
<td>uct_listener_params::conn_request_cb</td>
</tr>
<tr>
<td>UCT_LISTENER_PARAM_FIELD_USER_DATA</td>
<td>uct_listener_params::user_data</td>
</tr>
</tbody>
</table>
6.18.4.4 uct_cm_ep_priv_data_pack_args_field

enum uct_cm_ep_priv_data_pack_args_field

The enumeration allows specifying which fields in uct_cm_ep_priv_data_pack_args are present, for backward compatibility support.

**Enumerator**

<table>
<thead>
<tr>
<th>UCT_CM_EP_PRIV_DATA_PACK_ARGS_FIELD DEVICE_NAME</th>
<th>Enables uct_cm_ep_priv_data_pack_args::dev_name Indicates that dev_name field in uct_cm_ep_priv_data_pack_args_t is valid.</th>
</tr>
</thead>
</table>

6.18.4.5 uct_cm_remote_data_field

enum uct_cm_remote_data_field

The enumeration allows specifying which fields in uct_cm_remote_data are present, for backward compatibility support.

**Enumerator**

<table>
<thead>
<tr>
<th>UCT_CM_REMOTE_DATA_FIELD_DEV_ADDR</th>
<th>Enables uct_cm_remote_data::dev_addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCT_CM_REMOTE_DATA_FIELD_DEV_ADDR_LENGTH</td>
<td>Enables uct_cm_remote_data::dev_addr_length</td>
</tr>
<tr>
<td>UCT_CM_REMOTE_DATA_FIELD_CONN_PRIV_DATA</td>
<td>Enables uct_cm_remote_data::conn_priv_data</td>
</tr>
<tr>
<td>UCT_CM_REMOTE_DATA_FIELD_CONN_PRIV_DATA_LENGTH</td>
<td>Enables uct_cm_remote_data::conn_priv_data_length</td>
</tr>
</tbody>
</table>

6.18.4.6 uct_cm_listener_conn_request_args_field

enum uct_cm_listener_conn_request_args_field

The enumeration allows specifying which fields in uct_cm_listener_conn_request_args are present, for backward compatibility support.

**Enumerator**

<table>
<thead>
<tr>
<th>UCT_CM_LISTENER_CONN_REQUEST_ARGS_FIELD_DEV_NAME</th>
<th>Enables uct_cm_listener_conn_request_args::dev_name Indicates that dev_name field in uct_cm_listener_conn_request_args_t is valid.</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCT_CM_LISTENER_CONN_REQUEST_ARGS_FIELD_CONN_REQUEST</td>
<td>Enables uct_cm_listener_conn_request_args::conn_request Indicates that conn_request field in uct_cm_listener_conn_request_args_t is valid.</td>
</tr>
<tr>
<td>UCT_CM_LISTENER_CONN_REQUEST_ARGS_FIELD_REMOTE_DATA</td>
<td>Enables uct_cm_listener_conn_request_args::remote_data Indicates that remote_data field in uct_cm_listener_conn_request_args_t is valid.</td>
</tr>
</tbody>
</table>
### Enumerator

| UCT_CM_LISTENER_CONN_REQUEST_ARGS_FIELD_CLIENT_ADDR | Enables uct_cm_listener_conn_request_args::client_address
| Indicates that client_address field in uct_cm_listener_conn_request_args_t is valid. |

#### 6.18.4.7 uct_cm_ep_client_connect_args_field

```c
enum uct_cm_ep_client_connect_args_field
```

The enumeration allows specifying which fields in uct_cm_ep_client_connect_args are present, for backward compatibility support.

| UCT_CM_EP_CLIENT_CONNECT_ARGS_FIELD_REMOTE_DATA | Enables uct_cm_ep_client_connect_args::remote_data |
| UCT_CM_EP_CLIENT_CONNECT_ARGS_FIELD_STATUS | Enables uct_cm_ep_client_connect_args::status |

#### 6.18.4.8 uct_cm_ep_server_conn_notify_args_field

```c
enum uct_cm_ep_server_conn_notify_args_field
```

The enumeration allows specifying which fields in uct_cm_ep_server_conn_notify_args are present, for backward compatibility support.

| UCT_CM_EP_SERVER_CONN_NOTIFY_ARGS_FIELD_STATUS | Enables uct_cm_ep_server_conn_notify_args::status
| Indicates that status field in uct_cm_ep_server_conn_notify_args_t is valid. |

### 6.18.5 Function Documentation

#### 6.18.5.1 uct_iface_accept()

```c
ucs_status_t uct iface_accept (uct_iface_h iface, uct_conn_request_h conn_request )
```

**Parameters**

| in | iface | Transport interface which generated connection request conn_request. |
| in | conn_request | Connection establishment request passed as parameter of uct_sockaddr_conn_request_callback_t. |
Returns

Error code as defined by ucs_status_t

6.18.5.2 uct_iface_reject()

```c
ucs_status_t uct_iface_reject (  
    uct_iface_h iface,  
    uct_conn_request_h conn_request )
```

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>iface</th>
<th>Interface which generated connection establishment request conn_request.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>conn_request</td>
<td>Connection establishment request passed as parameter of uct_sockaddr_conn_request_callback_t.</td>
</tr>
</tbody>
</table>

Returns

Error code as defined by ucs_status_t

6.18.5.3 uct_ep_disconnect()

```c
ucs_status_t uct_ep_disconnect (  
    uct_ep_h ep,  
    unsigned flags )
```

This non-blocking routine will send a disconnect notification on the endpoint, so that uct_ep_disconnect_cb_t will be called on the remote peer. The remote side should also call this routine when handling the initiator's disconnect. After a call to this function, the given endpoint may not be used for communications anymore. The uct_ep_flush / uct_iface_flush routines will guarantee that the disconnect notification is delivered to the remote peer. uct_ep_destroy should be called on this endpoint after invoking this routine and uct_ep_params::disconnect_cb was called.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>ep</th>
<th>Endpoint to disconnect.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>flags</td>
<td>Reserved for future use.</td>
</tr>
</tbody>
</table>

Returns

UCS_OK Operation has completed successfully. UCS_ERR_BUSY The ep is not connected yet (either uct_cm_ep_client_connect_callback_t or uct_cm_ep_server_conn_notify_callback_t was not invoked). UCS_CINPROGRESS The disconnect request has been initiated, but the remote peer has not yet responded to this request, and consequently the registered callback uct_ep_disconnect_cb_t has not been invoked to handle the request. UCS_ERR_NOT_CONNECTED The ep is disconnected locally and remotely. Other error codes as defined by ucs_status_t.

6.18.5.4 uct_cm_open()

```c
ucs_status_t uct_cm_open (  
```

© 2021 Unified Communication X (UCX). All rights reserved.
Open a connection manager. All client server connection establishment operations are performed in the context of a specific connection manager.

Note
This is an alternative API for \texttt{uct\_iface\_open\_mode::UCT\_IFACE\_OPEN\_MODE\_SOCKADDR\_SERVER} and \texttt{uct\_iface\_open\_mode::UCT\_IFACE\_OPEN\_MODE\_SOCKADDR\_CLIENT}.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>component</th>
<th>Component on which to open the connection manager, as returned from \texttt{uct_query_components}.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>worker</td>
<td>Worker on which to open the connection manager.</td>
</tr>
<tr>
<td>in</td>
<td>config</td>
<td>CM configuration options. Either obtained from \texttt{uct_cm_config_read()} function, or pointer to CM-specific structure that extends \texttt{uct_cm_config_t}.</td>
</tr>
<tr>
<td>out</td>
<td>cm_p</td>
<td>Filled with a handle to the connection manager.</td>
</tr>
</tbody>
</table>

Returns

Error code.

6.18.5.5 \texttt{uct\_cm\_close()}

void \texttt{uct\_cm\_close} ( \texttt{uct\_cm\_h cm} )

Parameters

| in   | cm | Connection manager to close. |

6.18.5.6 \texttt{uct\_cm\_query()}

\texttt{ucs\_status\_t uct\_cm\_query (}
\texttt{     uct\_cm\_h cm,}
\texttt{     uct\_cm\_attr\_t \* cm\_attr) }

This routine queries the \texttt{cm} for its attributes \texttt{uct\_cm\_attr\_t}.

Parameters

| in   | cm | Connection manager to query. |
| out  | cm\_attr | Filled with connection manager attributes. |
6.18.5.7  uct_cm_config_read()

```c
uct_status_t uct_cm_config_read(
    uct_component_h component,
    const char * env_prefix,
    const char * filename,
    uct_cm_config_t ** config_p )
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>component</code></td>
<td>Read the configuration of the connection manager on this component.</td>
</tr>
<tr>
<td><code>env_prefix</code></td>
<td>If non-NULL, search for environment variables starting with this UCT_{&lt;prefix&gt;}. Otherwise, search for environment variables starting with just UCT_.</td>
</tr>
<tr>
<td><code>filename</code></td>
<td>If non-NULL, read configuration from this file. If the file does not exist, or exists but cannot be opened or read, it will be ignored.</td>
</tr>
<tr>
<td><code>config_p</code></td>
<td>Filled with a pointer to the configuration.</td>
</tr>
</tbody>
</table>

**Returns**

Error code.

6.18.5.8  uct_cm_client_ep_conn_notify()

```c
uct_status_t uct_cm_client_ep_conn_notify ( uct_ep_h ep )
```

This routine should be called on the client side after the client completed establishing its connection to the server. The routine will send a notification message to the server indicating that the client is connected.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ep</code></td>
<td>The connected endpoint on the client side.</td>
</tr>
</tbody>
</table>

**Returns**

Error code.

6.18.5.9  uct_listener_create()

```c
uct_status_t uct_listener_create ( uct_cm_h cm,
    const struct sockaddr * saddr,
    socklen_t socklen,
    const uct_listener_params_t * params,
    uct_listener_h * listener_p )
```

This routine creates a new listener on the given CM which will start listening on a given sockaddr.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cm</code></td>
<td>Connection manager on which to open the listener. This cm should not be closed as long as there are open listeners on it.</td>
</tr>
</tbody>
</table>
Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>saddr</code></td>
<td>The socket address to listen on.</td>
</tr>
<tr>
<td><code>socklen</code></td>
<td>The saddr length.</td>
</tr>
<tr>
<td><code>params</code></td>
<td>User defined <code>uct_listener_params_t</code> configurations for the <code>listener_p</code>.</td>
</tr>
<tr>
<td><code>listener_p</code></td>
<td>Filled with handle to the new listener.</td>
</tr>
</tbody>
</table>

Returns

Error code.

6.18.5.10  `uct_listener_destroy()`

```c
void uct_listener_destroy (  
    uct_listener_h listener  
)
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>listener</code></td>
<td>Listener to destroy.</td>
</tr>
</tbody>
</table>

6.18.5.11  `uct_listener_reject()`

```c
ucs_status_t uct_listener_reject (  
    uct_listener_h listener,  
    uct_conn_request_h conn_request  
)
```

This routine can be invoked on the server side. It rejects a connection request from the client.

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>listener</code></td>
<td>Listener which will reject the connection request.</td>
</tr>
<tr>
<td><code>conn_request</code></td>
<td>Connection establishment request passed as parameter of <code>uct_cm_listener_conn_request_callback_t</code> in <code>uct_cm_listener_conn_request_args_t::conn_request</code>.</td>
</tr>
</tbody>
</table>

Returns

Error code as defined by `ucs_status_t`.

6.18.5.12  `uct_listener_query()`

```c
ucs_status_t uct_listener_query (  
    uct_listener_h listener,  
    uct_listener_attr_t *listener_attr  
)
```

This routine queries the `listener` for its attributes `uct_listener_attr_t`. 

© 2021 Unified Communication X (UCX). All rights reserved.
### Parameters

<table>
<thead>
<tr>
<th></th>
<th><strong>listener</strong></th>
<th>Listener object to query.</th>
</tr>
</thead>
<tbody>
<tr>
<td>out</td>
<td><strong>listener_attr</strong></td>
<td>Filled with attributes of the listener.</td>
</tr>
</tbody>
</table>

### Returns

Error code as defined by `ucs_status_t`
6.19 UCT interface operations and capabilities

List of capabilities supported by UCX API.

Macros

- `#define UCT_IFACE_FLAG_AM_SHORT UCS_BIT(0)`
- `#define UCT_IFACE_FLAG_AM_BCOPY UCS_BIT(1)`
- `#define UCT_IFACE_FLAG_AM_ZCOPY UCS_BIT(2)`
- `#define UCT_IFACE_FLAG_PENDING UCS_BIT(3)`
- `#define UCT_IFACE_FLAG_PUT_SHORT UCS_BIT(4)`
- `#define UCT_IFACE_FLAG_PUT_BCOPY UCS_BIT(5)`
- `#define UCT_IFACE_FLAG_PUT_ZCOPY UCS_BIT(6)`
- `#define UCT_IFACE_FLAG_GET_SHORT UCS_BIT(8)`
- `#define UCT_IFACE_FLAG_GET_BCOPY UCS_BIT(9)`
- `#define UCT_IFACE_FLAG_GET_ZCOPY UCS_BIT(10)`
- `#define UCT_IFACE_FLAG_ATOMIC_CPU UCS_BIT(30)`
- `#define UCT_IFACE_FLAG_ATOMIC_DEVICE UCS_BIT(31)`
- `#define UCT_IFACE_FLAG_ERRHANDLE_SHORT_BUF UCS_BIT(32)`
- `#define UCT_IFACE_FLAG_ERRHANDLE_BCOPY_BUF UCS_BIT(33)`
- `#define UCT_IFACE_FLAG_ERRHANDLE_ZCOPY_BUF UCS_BIT(34)`
- `#define UCT_IFACE_FLAG_ERRHANDLE_AM_ID UCS_BIT(35)`
- `#define UCT_IFACE_FLAG_ERRHANDLE_REMOTE_MEM UCS_BIT(36)`
- `#define UCT_IFACE_FLAG_ERRHANDLE_BCOPY_LEN UCS_BIT(37)`
- `#define UCT_IFACE_FLAG_ERRHANDLE_PEER_FAILURE UCS_BIT(38)`
- `#define UCT_IFACE_FLAG_EP_CHECK UCS_BIT(39)`
- `#define UCT_IFACE_FLAG_CONNECT_TO_IFACE UCS_BIT(40)`
- `#define UCT_IFACE_FLAG_CONNECT_TO_EP UCS_BIT(41)`
- `#define UCT_IFACE_FLAG_CONNECT_TO_SOCKADDR UCS_BIT(42)`
- `#define UCT_IFACE_FLAG_AM_DUP UCS_BIT(43)`
- `#define UCT_IFACE_FLAG_CB_SYNC UCS_BIT(44)`
- `#define UCT_IFACE_FLAG_CB_ASYNC UCS_BIT(45)`
- `#define UCT_IFACE_FLAG_EP_KEEPALIVE UCS_BIT(46)`
- `#define UCT_IFACE_FLAG_TAG_EAGER_SHORT UCS_BIT(50)`
- `#define UCT_IFACE_FLAG_TAG_EAGER_BCOPY UCS_BIT(51)`
- `#define UCT_IFACE_FLAG_TAG_EAGER_ZCOPY UCS_BIT(52)`
- `#define UCT_IFACE_FLAG_TAG_RNDV_ZCOPY UCS_BIT(53)`

6.19.1 Detailed Description

The definition list presents a full list of operations and capabilities exposed by UCX API.

6.19.2 Macro Definition Documentation
6.19 UCT interface operations and capabilities

6.19.2.1 UCT_IFACE_FLAG_AM_SHORT

#define UCT_IFACE_FLAG_AM_SHORT UCS_BIT(0)
Short active message
Examples
   uct_hello_world.c.

6.19.2.2 UCT_IFACE_FLAG_AM_BCOPY

#define UCT_IFACE_FLAG_AM_BCOPY UCS_BIT(1)
Buffered active message
Examples
   uct_hello_world.c.

6.19.2.3 UCT_IFACE_FLAG_AM_ZCOPY

#define UCT_IFACE_FLAG_AM_ZCOPY UCS_BIT(2)
Zero-copy active message
Examples
   uct_hello_world.c.

6.19.2.4 UCT_IFACE_FLAG_PENDING

#define UCT_IFACE_FLAG_PENDING UCS_BIT(3)
Pending operations

6.19.2.5 UCT_IFACE_FLAG_PUT_SHORT

#define UCT_IFACE_FLAG_PUT_SHORT UCS_BIT(4)
Short put

6.19.2.6 UCT_IFACE_FLAG_PUT_BCOPY

#define UCT_IFACE_FLAG_PUT_BCOPY UCS_BIT(5)
Buffered put

6.19.2.7 UCT_IFACE_FLAG_PUT_ZCOPY

#define UCT_IFACE_FLAG_PUT_ZCOPY UCS_BIT(6)
Zero-copy put
6.19.2.8 UCT_IFACE_FLAG_GET_SHORT
#define UCT_IFACE_FLAG_GET_SHORT UCS_BIT(8)
Short get

6.19.2.9 UCT_IFACE_FLAG_GET_BCOPY
#define UCT_IFACE_FLAG_GET_BCOPY UCS_BIT(9)
Buffered get

6.19.2.10 UCT_IFACE_FLAG_GET_ZCOPY
#define UCT_IFACE_FLAG_GET_ZCOPY UCS_BIT(10)
Zero-copy get

6.19.2.11 UCT_IFACE_FLAG_ATOMIC_CPU
#define UCT_IFACE_FLAG_ATOMIC_CPU UCS_BIT(30)
Atomic communications are consistent with respect to CPU operations.

6.19.2.12 UCT_IFACE_FLAG_ATOMIC_DEVICE
#define UCT_IFACE_FLAG_ATOMIC_DEVICE UCS_BIT(31)
Atomic communications are consistent only with respect to other atomics on the same device.

6.19.2.13 UCT_IFACE_FLAG_ERRHANDLE_SHORT_BUF
#define UCT_IFACE_FLAG_ERRHANDLE_SHORT_BUF UCS_BIT(32)
Invalid buffer for short operation

6.19.2.14 UCT_IFACE_FLAG_ERRHANDLE_BCOPY_BUF
#define UCT_IFACE_FLAG_ERRHANDLE_BCOPY_BUF UCS_BIT(33)
Invalid buffer for buffered operation

6.19.2.15 UCT_IFACE_FLAG_ERRHANDLE_ZCOPY_BUF
#define UCT_IFACE_FLAG_ERRHANDLE_ZCOPY_BUF UCS_BIT(34)
Invalid buffer for zero copy operation

6.19.2.16 UCT_IFACE_FLAG_ERRHANDLE_AM_ID
#define UCT_IFACE_FLAG_ERRHANDLE_AM_ID UCS_BIT(35)
Invalid AM id on remote
6.19 UCT interface operations and capabilities

6.19.2.17 UCT_IFACE_FLAG_ERRHANDLE_REMOTE_MEM

#define UCT_IFACE_FLAG_ERRHANDLE_REMOTE_MEM UCS_BIT(36)

Remote memory access

6.19.2.18 UCT_IFACE_FLAG_ERRHANDLE_BCOPY_LEN

#define UCT_IFACE_FLAG_ERRHANDLE_BCOPY_LEN UCS_BIT(37)

Invalid length for buffered operation

6.19.2.19 UCT_IFACE_FLAG_ERRHANDLE_PEER_FAILURE

#define UCT_IFACE_FLAG_ERRHANDLE_PEER_FAILURE UCS_BIT(38)

Remote peer failures/outage

6.19.2.20 UCT_IFACE_FLAG_EP_CHECK

#define UCT_IFACE_FLAG_EP_CHECK UCS_BIT(39)

Endpoint check

6.19.2.21 UCT_IFACE_FLAG_CONNECT_TO_IFACE

#define UCT_IFACE_FLAG_CONNECT_TO_IFACE UCS_BIT(40)

Supports connecting to interface

Examples

    uct_hello_world.c.

6.19.2.22 UCT_IFACE_FLAG_CONNECT_TO_EP

#define UCT_IFACE_FLAG_CONNECT_TO_EP UCS_BIT(41)

Supports connecting to specific endpoint

Examples

    uct_hello_world.c.

6.19.2.23 UCT_IFACE_FLAG_CONNECT_TO_SOCKADDR

#define UCT_IFACE_FLAG_CONNECT_TO_SOCKADDR UCS_BIT(42)

Supports connecting to sockaddr
6.19.2.24 UCT_IFACE_FLAG_AM_DUP

#define UCT_IFACE_FLAG_AM_DUP UCS_BIT(43)

Active messages may be received with duplicates. This happens if the transport does not keep enough information to detect retransmissions.

6.19.2.25 UCT_IFACE_FLAG_CB_SYNC

#define UCT_IFACE_FLAG_CB_SYNC UCS_BIT(44)

Interface supports setting a callback which is invoked only from the calling context of uct_worker_progress().

6.19.2.26 UCT_IFACE_FLAG_CB_ASYNC

#define UCT_IFACE_FLAG_CB_ASYNC UCS_BIT(45)

Interface supports setting a callback which will be invoked within a reasonable amount of time if uct_worker_progress() is not being called. The callback can be invoked from any progress context and it may also be invoked when uct_worker_progress() is called.

6.19.2.27 UCT_IFACE_FLAG_EP_KEEPALIVE

#define UCT_IFACE_FLAG_EP_KEEPALIVE UCS_BIT(46)

Transport endpoint has built-in keepalive feature, which guarantees the error callback on the transport interface will be called if the communication channel with remote peer is broken, even if there are no outstanding send operations.

6.19.2.28 UCT_IFACE_FLAG_TAG_EAGER_SHORT

#define UCT_IFACE_FLAG_TAG_EAGER_SHORT UCS_BIT(50)

Hardware tag matching short eager support.

6.19.2.29 UCT_IFACE_FLAG_TAG_EAGER_BCOPY

#define UCT_IFACE_FLAG_TAG_EAGER_BCOPY UCS_BIT(51)

Hardware tag matching bcopy eager support.

6.19.2.30 UCT_IFACE_FLAG_TAG_EAGER_ZCOPY

#define UCT_IFACE_FLAG_TAG_EAGER_ZCOPY UCS_BIT(52)

Hardware tag matching zcopy eager support.

6.19.2.31 UCT_IFACE_FLAG_TAG_RNDV_ZCOPY

#define UCT_IFACE_FLAG_TAG_RNDV_ZCOPY UCS_BIT(53)

Hardware tag matching rendezvous zcopy support.
6.20 UCT interface for asynchronous event capabilities

List of capabilities supported by UCT iface event API.

Macros

- `#define UCT_IFACE_FLAG_EVENT_SEND_COMP UCS_BIT(0)`
- `#define UCT_IFACE_FLAG_EVENT_RECV UCS_BIT(1)`
- `#define UCT_IFACE_FLAG_EVENT_RECV_SIG UCS_BIT(2)`
- `#define UCT_IFACE_FLAG_EVENT_FD UCS_BIT(3)`
- `#define UCT_IFACE_FLAG_EVENT_ASYNC_CB UCS_BIT(4)`

6.20.1 Detailed Description

The definition list presents a full list of operations and capabilities supported by UCT iface event.

6.20.2 Macro Definition Documentation

6.20.2.1 UCT_IFACE_FLAG_EVENT_SEND_COMP

`#define UCT_IFACE_FLAG_EVENT_SEND_COMP UCS_BIT(0)`
Event notification of send completion is supported

6.20.2.2 UCT_IFACE_FLAG_EVENT_RECV

`#define UCT_IFACE_FLAG_EVENT_RECV UCS_BIT(1)`
Event notification of tag and active message receive is supported

6.20.2.3 UCT_IFACE_FLAG_EVENT_RECV_SIG

`#define UCT_IFACE_FLAG_EVENT_RECV_SIG UCS_BIT(2)`
Event notification of signaled tag and active message is supported

6.20.2.4 UCT_IFACE_FLAG_EVENT_FD

`#define UCT_IFACE_FLAG_EVENT_FD UCS_BIT(3)`
Event notification through File Descriptor is supported

6.20.2.5 UCT_IFACE_FLAG_EVENT_ASYNC_CB

`#define UCT_IFACE_FLAG_EVENT_ASYNC_CB UCS_BIT(4)`
Event notification through asynchronous callback invocation is supported
6.21 Unified Communication Services (UCS) API

Modules

- UCS Communication Resource

6.21.1 Detailed Description

This section describes UCS API.
6.22 UCS Communication Resource

Data Structures

- struct ucs_sock_addr

Typedefs

- typedef void(\∗ucs_async_event_cb_t) (int id, ucs_event_set_types_t events, void ∗arg)
- typedef struct ucs_sock_addr ucs_sock_addr_t
- typedef enum ucs_memory_type ucs_memory_type_t
  Memory types.
- typedef unsigned long ucs_time_t
- typedef void ∗ucs_status_ptr_t
  Status pointer.

Enumerations

- enum ucs_callbackq_flags { UCS_CALLBACKQ_FLAG_FAST = UCS_BIT(0), UCS_CALLBACKQ_FLAG_ONESHOT = UCS_BIT(1) }
- enum ucs_memory_type {
  UCS_MEMORY_TYPE_HOST, UCS_MEMORY_TYPE_CUDA, UCS_MEMORY_TYPE_CUDA_MANAGED,
  UCS_MEMORY_TYPE_ROCM,
  UCS_MEMORY_TYPE_ROCM_MANAGED, UCS_MEMORY_TYPE_LAST, UCS_MEMORY_TYPE_UNKNOWN = UCS_MEMORY_TYPE_LAST }
  Memory types.
- enum ucs_status_t {
  UCS_OK = 0, UCS_INPROGRESS = 1, UCS_ERR_NO_MESSAGE = -1, UCS_ERR_NO_RESOURCE = -2,
  UCS_ERR_IO_ERROR = -3, UCS_ERR_NO_MEMORY = -4, UCS_ERR_INVALID_PARAM = -5,
  UCS_ERR_UNREACHABLE = -6,
  UCS_ERR_INVALID_ADDR = -7, UCS_ERR_NOT_IMPLEMENTED = -8, UCS_ERR_MESSAGE_TRUNCATED = -9,
  UCS_ERR_NO_PROGRESS = -10,
  UCS_ERR_BUFFER_TOO_SMALL = -11, UCS_ERR_NO_ELEM = -12, UCS_ERR_SOME_CONNECTS_FAILED = -13,
  UCS_ERR_NO_DEVICE = -14,
  UCS_ERR_BUSY = -15, UCS_ERR_CANCELED = -16, UCS_ERR_SHMEM_SEGMENT = -17,
  UCS_ERR_ALREADY_EXISTS = -18,
  UCS_ERR_OUT_OF_RANGE = -19, UCS_ERR_TIMED_OUT = -20, UCS_ERR_EXCEEDS_LIMIT = -21,
  UCS_ERR_UNSUPPORTED = -22,
  UCS_ERR_REJECTED = -23, UCS_ERR_NOT_CONNECTED = -24, UCS_ERR_CONNECTION_RESET = -25,
  UCS_ERR_FIRST_LINK_FAILURE = -40,
  UCS_ERR_LAST_LINK_FAILURE = -59, UCS_ERR_FIRST_ENDPOINT_FAILURE = -60, UCS_ERR_ENDPOINT_TIMEOUT = -80,
  UCS_ERR_LAST_ENDPOINT_FAILURE = -89,
  UCS_ERR_LAST = -100 }
  Status codes.
- enum ucs_thread_mode_t { UCS_THREAD_MODE_SINGLE, UCS_THREAD_MODE_SERIALIZED,
  UCS_THREAD_MODE_MULTI, UCS_THREAD_MODE_LAST }
  Thread sharing mode.

Functions

- ucs_status_t ucs_async_set_event_handler (ucs_async_mode_t mode, int event_fd, ucs_event_set_types_t events, ucs_async_event_cb_t cb, void ∗arg, ucs_async_context_t ∗async)
• ucs_status_t ucs_async_add_timer (ucs_async_mode_t mode, ucs_time_t interval, ucs_async_event_cb_t cb, void *arg, ucs_async_context_t *async, int *timer_id_p)
• ucs_status_t ucs_async_remove_handler (int id, int sync)
• ucs_status_t ucs_async_modify_handler (int fd, ucs_event_set_types_t events)
• ucs_status_t ucs_async_context_create (ucs_async_mode_t mode, ucs_async_context_t **async_p)

Create an asynchronous execution context.

• void ucs_async_context_destroy (ucs_async_context_t *async)

Destroy the asynchronous execution context.

• void ucs_async_poll (ucs_async_context_t *async)

6.22.1 Detailed Description

This section describes a concept of the Communication Resource and routines associated with the concept.

6.22.2 Data Structure Documentation

6.22.2.1 struct ucs_sock_addr

BSD socket address specification.

Data Fields

<table>
<thead>
<tr>
<th>const struct sockaddr *</th>
<th>addr</th>
<th>Pointer to socket address</th>
</tr>
</thead>
<tbody>
<tr>
<td>socklen_t</td>
<td>addrlen</td>
<td>Address length</td>
</tr>
</tbody>
</table>

6.22.3 Typedef Documentation

6.22.3.1 ucs_async_event_cb_t

typedef void(* ucs_async_event_cb_t) (int id, ucs_event_set_types_t events, void *arg)

Async event callback.

Parameters

<table>
<thead>
<tr>
<th>id</th>
<th>Event id (timer or file descriptor).</th>
</tr>
</thead>
<tbody>
<tr>
<td>events</td>
<td>The events that triggered the callback.</td>
</tr>
<tr>
<td>arg</td>
<td>User-defined argument.</td>
</tr>
</tbody>
</table>

6.22.3.2 ucs_sock_addr_t

typedef struct ucs_sock_addr ucs_sock_addr_t

BSD socket address specification.
6.22.3.3 ucs_memory_type_t

typedef enum ucs_memory_type ucs_memory_type_t

List of supported memory types.

6.22.3.4 ucs_time_t

typedef unsigned long ucs_time_t

UCS time units. These are not necessarily aligned with metric time units. MUST compare short time values with UCS_SHORT_TIME_CMP to handle wrap-around.

6.22.3.5 ucs_status_ptr_t

typedef void* ucs_status_ptr_t

A pointer can represent one of these values:

- NULL / UCS_OK
- Error code pointer (UCS_ERR_xx)
- Valid pointer

6.22.4 Enumeration Type Documentation

6.22.4.1 ucs_callbackq_flags

enum ucs_callbackq_flags

Callback flags

Enumeration

<table>
<thead>
<tr>
<th>Enumerators</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCS_CALLBACKQ_FLAG_FAST</td>
<td>Fast-path (best effort)</td>
</tr>
<tr>
<td>UCS_CALLBACKQ_FLAG_ONESHOT</td>
<td>Call the callback only once (cannot be used with FAST)</td>
</tr>
</tbody>
</table>

6.22.4.2 ucs_memory_type

enum ucs_memory_type

List of supported memory types.

Enumeration

<table>
<thead>
<tr>
<th>Enumerators</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCS_MEMORY_TYPE_HOST</td>
<td>Default system memory</td>
</tr>
<tr>
<td>UCS_MEMORY_TYPE_CUDA</td>
<td>NVIDIA CUDA memory</td>
</tr>
<tr>
<td>UCS_MEMORY_TYPE_CUDA_MANAGED</td>
<td>NVIDIA CUDA managed (or unified) memory</td>
</tr>
<tr>
<td>UCS_MEMORY_TYPE_ROCM</td>
<td>AMD ROCM memory</td>
</tr>
<tr>
<td>UCS_MEMORY_TYPE_ROCM_MANAGED</td>
<td>AMD ROCM managed system memory</td>
</tr>
<tr>
<td>UCS_MEMORY_TYPE_LAST</td>
<td></td>
</tr>
</tbody>
</table>
### Enumerator

<table>
<thead>
<tr>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCS_MEMORY_TYPE_UNKNOWN</td>
</tr>
</tbody>
</table>

### 6.22.4.3 ucs_status_t

**enum ucs_status_t**

**Note**

In order to evaluate the necessary steps to recover from a certain error, all error codes which can be returned by the external API are grouped by the largest entity permanently effected by the error. Each group ranges between its UCS_ERR_FIRST_{name} and UCS_ERR_LAST_{name} enum values. For example, if a link fails it may be sufficient to destroy (and possibly replace) it, in contrast to an endpoint-level error.

**Enumerator**

<table>
<thead>
<tr>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCS_OK</td>
</tr>
<tr>
<td>UCS_INPROGRESS</td>
</tr>
<tr>
<td>UCS_ERR_NO_MESSAGE</td>
</tr>
<tr>
<td>UCS_ERR_NO_RESOURCE</td>
</tr>
<tr>
<td>UCS_ERR_IO_ERROR</td>
</tr>
<tr>
<td>UCS_ERR_NO_MEMORY</td>
</tr>
<tr>
<td>UCS_ERR_INVALID_PARAM</td>
</tr>
<tr>
<td>UCS_ERR_UNREACHABLE</td>
</tr>
<tr>
<td>UCS_ERR_INVALID_ADDR</td>
</tr>
<tr>
<td>UCS_ERR_NOT_IMPLEMENTED</td>
</tr>
<tr>
<td>UCS_ERR_MESSAGE_TRUNCATED</td>
</tr>
<tr>
<td>UCS_ERR_NO_PROGRESS</td>
</tr>
<tr>
<td>UCS_ERR_BUFFER_TOO_SMALL</td>
</tr>
<tr>
<td>UCS_ERR_NO_ELEM</td>
</tr>
<tr>
<td>UCS_ERR_SOME_CONNECTS_FAILED</td>
</tr>
<tr>
<td>UCS_ERR_NO_DEVICE</td>
</tr>
<tr>
<td>UCS_ERR_BUSY</td>
</tr>
<tr>
<td>UCS_ERR_CANCELED</td>
</tr>
<tr>
<td>UCS_ERR_SHMEM_SEGMENT</td>
</tr>
<tr>
<td>UCS_ERR_ALREADY_EXISTS</td>
</tr>
<tr>
<td>UCS_ERR_OUT_OF_RANGE</td>
</tr>
<tr>
<td>UCS_ERR_TIMED_OUT</td>
</tr>
<tr>
<td>UCS_ERR_EXCEEDS_LIMIT</td>
</tr>
<tr>
<td>UCS_ERR_UNSUPPORTED</td>
</tr>
<tr>
<td>UCS_ERR_REJECTED</td>
</tr>
<tr>
<td>UCS_ERR_NOT_CONNECTED</td>
</tr>
<tr>
<td>UCS_ERR_CONNECTION_RESET</td>
</tr>
<tr>
<td>UCS_ERR_FIRST_LINK_FAILURE</td>
</tr>
<tr>
<td>UCS_ERR_LAST_LINK_FAILURE</td>
</tr>
<tr>
<td>UCS_ERR_FIRST_ENDPOINT_FAILURE</td>
</tr>
<tr>
<td>UCS_ERR_ENDPOINT_TIMEOUT</td>
</tr>
<tr>
<td>UCS_ERR_LAST_ENDPOINT_FAILURE</td>
</tr>
<tr>
<td>UCS_ERR_LAST</td>
</tr>
</tbody>
</table>
6.22.4.4 ucs_thread_mode_t

enum ucs_thread_mode_t

Specifies thread sharing mode of an object.

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCS_THREAD_MODE_SINGLE</td>
<td>Only the master thread can access (i.e. the thread that initialized the context; multiple threads may exist and never access)</td>
</tr>
<tr>
<td>UCS_THREAD_MODE_SERIALIZED</td>
<td>Multiple threads can access, but only one at a time</td>
</tr>
<tr>
<td>UCS_THREAD_MODE_MULTI</td>
<td>Multiple threads can access concurrently</td>
</tr>
</tbody>
</table>

6.22.5 Function Documentation

6.22.5.1 ucs_async_set_event_handler()

```c
ucs_status_t ucs_async_set_event_handler (
    ucs_async_mode_t mode,
    int event_fd,
    ucs_event_set_types_t events,
    ucs_async_event_cb_t cb,
    void * arg,
    ucs_async_context_t * async )
```

Register a file descriptor for monitoring (call handler upon events). Every fd can have only one handler.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode</td>
<td>Thread or signal.</td>
</tr>
<tr>
<td>event_fd</td>
<td>File descriptor to set handler for.</td>
</tr>
<tr>
<td>events</td>
<td>Events to wait on (UCS_EVENT_SET_EVxxx bits).</td>
</tr>
<tr>
<td>cb</td>
<td>Callback function to execute.</td>
</tr>
<tr>
<td>arg</td>
<td>Argument to callback.</td>
</tr>
<tr>
<td>async</td>
<td>Async context to which events are delivered. If NULL, safety is up to the user.</td>
</tr>
</tbody>
</table>

Returns

Error code as defined by `ucs_status_t`. 
6.22.5.2  

```c
ucc_status_t ucs_async_add_timer(
    ucs_async_mode_t mode,
    ucs_time_t interval,
    ucs_async_event_cb_t cb,
    void * arg,
    ucs_async_context_t * async,
    int * timer_id_p )
```

Add timer handler.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mode</code></td>
<td>Thread or signal.</td>
</tr>
<tr>
<td><code>interval</code></td>
<td>Timer interval.</td>
</tr>
<tr>
<td><code>cb</code></td>
<td>Callback function to execute.</td>
</tr>
<tr>
<td><code>arg</code></td>
<td>Argument to callback.</td>
</tr>
<tr>
<td><code>async</code></td>
<td>Async context to which events are delivered. If NULL, safety is up to the user.</td>
</tr>
<tr>
<td><code>timer_id_p</code></td>
<td>Filled with timer id.</td>
</tr>
</tbody>
</table>

**Returns**

Error code as defined by `ucc_status_t`.

6.22.5.3  

```c
ucc_status_t ucs_async_remove_handler(
    int id,
    int sync )
```

Remove an event handler (Timer or event file).

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>id</code></td>
<td>Timer/FD to remove.</td>
</tr>
<tr>
<td><code>sync</code></td>
<td>If nonzero, wait until the handler for this event is not running anymore. If called from the context of the callback, the handler will be removed immediately after the current callback returns.</td>
</tr>
</tbody>
</table>

**Returns**

Error code as defined by `ucc_status_t`.

6.22.5.4  

```c
ucc_status_t ucs_async_modify_handler(
    int fd,
    ucs_event_set_types_t events )
```

Modify events mask for an existing event handler (event file).
Parameters

<table>
<thead>
<tr>
<th>fd</th>
<th>File descriptor modify events for.</th>
</tr>
</thead>
<tbody>
<tr>
<td>events</td>
<td>New set of events to wait on (UCS_EVENT_SET_EVxxx bits).</td>
</tr>
</tbody>
</table>

Returns

Error code as defined by ucs_status_t.

6.22.5.5 ucs_async_context_create()

```c
ucs_status_t ucs_async_context_create (  
    ucs_async_mode_t mode,  
    ucs_async_context_t ** async_p )
```
Allocate and initialize an asynchronous execution context. This can be used to ensure safe event delivery.

Parameters

<table>
<thead>
<tr>
<th>mode</th>
<th>Indicates whether to use signals or polling threads for waiting.</th>
</tr>
</thead>
<tbody>
<tr>
<td>async_p</td>
<td>Event context pointer to initialize.</td>
</tr>
</tbody>
</table>

Returns

Error code as defined by ucs_status_t.

Examples

uct_hello_world.c.

6.22.5.6 ucs_async_context_destroy()

```c
void ucs_async_context_destroy (  
    ucs_async_context_t * async )
```
Clean up the async context, and release system resources if possible. The context memory released.

Parameters

| async | Asynchronous context to clean up. |

Examples

uct_hello_world.c.
6.22.5.7 ucs_async_poll()

void ucs_async_poll (  
    ucs_async_context_t * async )

Poll on async context.

Parameters

| async | Async context to poll on. NULL polls on all. |
Chapter 7

Data Structure Documentation

7.1 ucp_generic_dt_ops Struct Reference

UCP generic data type descriptor.

Data Fields

- void *(start_pack)(void *context, const void *buffer, size_t count)
  
  Start a packing request.

- void *(start_unpack)(void *context, void *buffer, size_t count)
  
  Start an unpacking request.

- size_t *(packed_size)(void *state)
  
  Get the total size of packed data.

- size_t *(pack)(void *state, size_t offset, void *dest, size_t max_length)
  
  Pack data.

- ucs_status_t *(unpack)(void *state, size_t offset, const void *src, size_t length)
  
  Unpack data.

- void *(finish)(void *state)
  
  Finish packing/unpacking.

7.1.1 Detailed Description

This structure provides a generic datatype descriptor that is used for definition of application defined datatypes.

Typically, the descriptor is used for an integration with datatype engines implemented within MPI and SHMEM implementations.

Note

In case of partial receive, any amount of received data is acceptable which matches buffer size.

The documentation for this struct was generated from the following file:

- ucp.h

7.2 uct_tag_context Struct Reference

Posted tag context.
Data Fields

- void(* tag_consumed_cb )(uct_tag_context_t *self)
- void(* completed_cb )(uct_tag_context_t *self, uct_tag_t stag, uint64_t imm, size_t length, void *inline_data, ucs_status_t status)
- void(* rndv_cb )(uct_tag_context_t *self, uct_tag_t stag, const void *header, unsigned header_length, ucs_status_t status, unsigned flags)
- char priv [UCT_TAG_PRIV_LEN]

7.2.1 Detailed Description

Tag context is an object which tracks a tag posted to the transport. It contains callbacks for matching events on this tag.

7.2.2 Field Documentation

7.2.2.1 tag_consumed_cb

void(* uct_tag_context::tag_consumed_cb) (uct_tag_context_t *self)

Tag is consumed by the transport and should not be matched in software.

Parameters

| in | self | Pointer to relevant context structure, which was initially passed to uct_iface_tag_recv_zcopy. |

7.2.2.2 completed_cb

void(* uct_tag_context::completed_cb) (uct_tag_context_t *self, uct_tag_t stag, uint64_t imm, size_t length, void *inline_data, ucs_status_t status)

Tag processing is completed by the transport.

Parameters

| in | self | Pointer to relevant context structure, which was initially passed to uct_iface_tag_recv_zcopy. |
| in | stag | Tag from sender. |
| in | imm | Immediate data from sender. For rendezvous, it's always 0. |
| in | length | Completed length. |
| in | inline_data | If non-null, points to a temporary buffer which contains the received data. In this case the received data was not placed directly in the receive buffer. This callback routine is responsible for copy-out the inline data, otherwise it is released. |
| in | status | Completion status: (a) UCS_OK - Success, data placed in provided buffer. (b) UCS_ERR_TRUNCATED - Sender's length exceed posted buffer, no data is copied. (c) UCS_ERR_CANCELED - Canceled by user. |
7.2.2.3 rndv_cb

```c
void(* uct_tag_context::rndv_cb) (uct_tag_context_t *self, uct_tag_t stag, const void *header,
unsigned header_length, ucs_status_t status, unsigned flags)
```

Tag was matched by a rendezvous request, which should be completed by the protocol layer.

### Parameters

<table>
<thead>
<tr>
<th>In</th>
<th>self</th>
<th>Pointer to relevant context structure, which was initially passed to uct_iface_tag_recv_zcopy.</th>
</tr>
</thead>
<tbody>
<tr>
<td>In</td>
<td>stag</td>
<td>Tag from sender.</td>
</tr>
<tr>
<td>In</td>
<td>header</td>
<td>User defined header.</td>
</tr>
<tr>
<td>In</td>
<td>header_length</td>
<td>User defined header length in bytes.</td>
</tr>
<tr>
<td>In</td>
<td>status</td>
<td>Completion status.</td>
</tr>
<tr>
<td>In</td>
<td>flags</td>
<td>Flags defined by UCT_TAG_RECV_CB_xx.</td>
</tr>
</tbody>
</table>

7.2.2.4 priv

```c
char uct_tag_context::priv[UCT_TAG_PRIV_LEN]
```

A placeholder for the private data used by the transport

The documentation for this struct was generated from the following file:

- uct.h
Chapter 8

Example Documentation

8.1 ucp_client_server.c

UCP client / server example using different APIs (tag, stream, am) utility.

/*
 * UCP client - server example utility
 * -------------------------------
 * Server side:
 * ./ucp_client_server
 * Client side:
 * ./ucp_client_server -a <server-ip>
 * Notes:
 * - The server will listen to incoming connection requests on INADDR_ANY.
 * - The client needs to pass the IP address of the server side to connect to
 * as an argument to the test.
 * - Currently, the passed IP needs to be an IPoIB or a RoCE address.
 * - The port which the server side would listen on can be modified with the
 *   '-p' option and should be used on both sides. The default port to use is
 *   13337.
*/
#include <ucp/api/ucp.h>
#include <string.h> /* memset */
#include <arpa/inet.h> /* inet_addr */
#include <unistd.h> /* getopt */
#include <stdlib.h> /* atoi */
#define TEST_STRING_LEN sizeof(test_message)
#define DEFAULT_PORT 13337
#define IP_STRING_LEN 50
#define PORT_STRING_LEN 8
#define TAG 0xCAFEd
#define COMM_TYPE_DEFAULT "STREAM"
#define PRINT_INTERVAL 2000
#define DEFAULT_NUM_ITERATIONS 1
#define TEST_AM_ID 0
const char test_message[] = "UCX Client-Server Hello World";
static uint16_t server_port = DEFAULT_PORT;
static int num_iterations = DEFAULT_NUM_ITERATIONS;
typedef enum {
    CLIENT_SERVER_SEND_RECV_STREAM = UCS_BIT(0),
    CLIENT_SERVER_SEND_RECV_TAG = UCS_BIT(1),
    CLIENT_SERVER_SEND_RECV_AM = UCS_BIT(2),
} send_recv_type_t;
typedef struct ucx_server_ctx {
    volatile ucp_conn_request_h conn_request;
    ucp_listener_h listener;
} ucx_server_ctx_t;
typedef struct test_req {
    int complete;
} test_req_t;
__attribute__((aligned))
typedef struct ucx_client_request {
    volatile ucp_conn_request_h conn_request;
    ucp_listener_h listener;
} ucx_client_request_t;
typedef struct ucx_server_ctx_t;
typedef struct test_req_t {
    int complete;
} test_req_t;
static struct {
    volatile int complete;
    int is_rndv;
    void *desc;
    void *recv_buf;
} am_data_desc = {0, 0, NULL, NULL};
static void usage(void);
static void tag_recv_cb(void *request, ucs_status_t status,
    const ucp_tag_recv_info_t *info, void *user_data)
{
    test_req_t *ctx = user_data;
    ctx->complete = 1;
}
static void stream_recv_cb(void *request, ucs_status_t status, size_t length,
    void *user_data)
{
    test_req_t *ctx = user_data;
    ctx->complete = 1;
}
static void am_recv_cb(void *request, ucs_status_t status, size_t length,
    void *user_data)
{
    test_req_t *ctx = user_data;
    ctx->complete = 1;
}
static void send_cb(void *request, ucs_status_t status, void *user_data)
{
    test_req_t *ctx = user_data;
    ctx->complete = 1;
}
static void err_cb(void *arg, ucp_ep_h ep, ucs_status_t status)
{
    printf("error handling callback was invoked with status %d (%s)\n", status, ucs_status_string(status));
}
void set_listen_addr(const char *address_str, struct sockaddr_in *listen_addr)
{
    /* The server will listen on INADDR_ANY */
    memset(listen_addr, 0, sizeof(struct sockaddr_in));
    listen_addr->sin_family = AF_INET;
    listen_addr->sin_addr.s_addr = (address_str) ? inet_addr(address_str) : INADDR_ANY;
    listen_addr->sin_port = htons(server_port);
}
void set_connect_addr(const char *address_str, struct sockaddr_in *connect_addr)
{
    memset(connect_addr, 0, sizeof(struct sockaddr_in));
    connect_addr->sin_family = AF_INET;
    connect_addr->sin_addr.s_addr = inet_addr(address_str);
    connect_addr->sin_port = htons(server_port);
}
static ucs_status_t start_client(ucp_worker_h ucp_worker, const char *ip,
    ucp_ep_h *client_ep)
{
    ucp_ep_params_t ep_params;
    struct sockaddr_in connect_addr;
    ucs_status_t status;
    set_connect_addr(ip, &connect_addr);
    /* Endpoint field mask bits:
     * UCP_EP_PARAM_FIELD_FLAGS - Use the value of the 'flags' field.
     * UCP_EP_PARAM_FIELD_SOCK_ADDR - Use a remote sockaddr to connect
     *     to the remote peer.
     * UCP_EP_PARAM_FIELD_ERR_HANDLING_MODE - Error handling mode - this flag
     * is temporarily required since the
     *     endpoint will be closed with
     *     UCP_EP_CLOSE_MODE_FORCE which
     *     requires this mode.
     *     Once UCP_EP_CLOSE_MODE_FORCE is
     *     removed, the error handling mode
     *     will be removed.
     */
    ep_params.err_mode = UCP_ERR_HANDLING_MODE_PEER;
    ep_params.err_handler.cb = err_cb;
    ep_params.err_handler.arg = NULL;
    ep_params.sockaddr.addr = (struct sockaddr *)&connect_addr;
    ep_params.sockaddr.addrlen = sizeof(connect_addr);
    status = ucp_ep_create(ucp_worker, &ep_params, client_ep);
    if (status != UCS_OK) {
        fprintf(stderr, "failed to connect to %s (%s)\n", ip, ucs_status_string(status));
        return status;
    }
    return status;
}
static void print_result(int is_server, char *recv_message, int current_iter)
{
    if (is_server) {
        printf("Server: iteration #%d\n", (current_iter + 1));
        printf("UCX data message was received\n");
    }
}

© 2021 Unified Communication X (UCX). All rights reserved.
```c
printf("\n--- UCP TEST SUCCESS \n\n");
printf("\n-------------------\n\n");
}

if (request == NULL) {
    return UCS_OK;
}
if (UCS_PTR_IS_ERR(request)) {
    return UCS_PTR_STATUS(request);
}
while (ctx->complete == 0) {
    ucp_worker_progress(ucp_worker);
}
status = ucp_request_check_status(request);
ucp_request_free(request);
return status;
}

int send_recv_stream(ucp_worker_h ucp_worker, ucp_ep_h ep, int is_server,
int current_iter) {
    char recv_message[TEST_STRING_LEN]= "";
    ucp_request_param_t param;
test_req_t *request;
    size_t length;
test_req_t ctx;
tx.complete = 0;
    param.op_attr_mask = UCP_OP_ATTR_FIELD_CALLBACK |
                         UCP_OP_ATTR_FIELD_USER_DATA;
    param.user_data = &ctx;
if (!is_server) {
    /* Client sends a message to the server using the stream API */
    param.cb.send = send_cb;
    request = ucp_stream_send_nbx(ep, test_message, TEST_STRING_LEN,
                              &param);
} else {
    /* Server receives a message from the client using the stream API */
    param.cb.recv_stream = stream_recv_cb;
    request = ucp_stream_recv_nbx(ep, &recv_message, TEST_STRING_LEN,
                              &length, &param);
}
return request_finalize(ucp_worker, request, &ctx, is_server, recv_message, current_iter);
}

int send_recv_tag(ucp_worker_h ucp_worker, ucp_ep_h ep, int is_server,
int current_iter) {
    char recv_message[TEST_STRING_LEN]= "";
    ucp_request_param_t param;
test_req_t *request;
    size_t length;
test_req_t ctx;
tx.complete = 0;
    param.op_attr_mask = UCP_OP_ATTR_FIELD_CALLBACK |
                         UCP_OP_ATTR_FIELD_USER_DATA;
    param.user_data = &ctx;
if (!is_server) {
    /* Client sends a message to the server using the stream API */
    param.cb.send = send_cb;
    request = ucp_stream_send_nbx(ep, test_message, TEST_STRING_LEN,
                              &param);
} else {
    /* Server receives a message from the client using the stream API */
    param.cb.recv_stream = stream_recv_cb;
    request = ucp_stream_recv_nbx(ep, &recv_message, TEST_STRING_LEN,
                              &length, &param);
}
return request_finalize(ucp_worker, request, &ctx, is_server, recv_message, current_iter);
}
```
param.op_attr_mask = UCP_OP_ATTR_FIELD_CALLBACK | UCP_OP_ATTR_FIELD_USER_DATA;
param.user_data = &ctx;
if (!is_server) {
  /* Client sends a message to the server using the Tag-Matching API */
  param.cb.send = send_cb;
  request = ucp_tag_send_nbx(ep, test_message, TEST_STRING_LEN, TAG, &param);
} else {
  /* Server receives a message from the client using the Tag-Matching API */
  param.cb.recv = tag_recv_cb;
  request = ucp_tag_recv_nbx(ucp_worker, &recv_message, TEST_STRING_LEN, TAG, 0, &param);
}
return request_finalize(ucp_worker, request, &ctx, is_server, recv_message, current_iter);
}

ucs_status_t ucp_am_data_cb(void *arg, const void *header, size_t header_length, void *data, size_t length, const ucp_am_recv_param_t *param) {
if (length != TEST_STRING_LEN) {
  fprintf(stderr, "received wrong data length %ld (expected %ld)", length, TEST_STRING_LEN);
  goto out;
}
if ((header != NULL) || (header_length != 0)) {
  fprintf(stderr, "received unexpected header, length %ld", header_length);
}
if (param->recv_attr & UCP_AM_RECV_ATTR_FLAG_RNDV) {
  /* Rendezvous request arrived, data contains an internal UCX descriptor, which has to be passed to ucp_am_recv_data_nbx function to confirm data transfer. */
  am_data_desc.is_rndv = 1;
  am_data_desc.desc = data;
  return UCS_INPROGRESS;
}
/* Message delivered with eager protocol, data should be available immediately */
am_data_desc.is_rndv = 0;
memcpy(am_data_desc.recv_buf, data, length);
out:
am_data_desc.complete = 1;
return UCS_OK;
}

static int send_recv_am(ucp_worker_h ucp_worker, ucp_ep_h ep, int is_server, int current_iter) {
char recv_message[TEST_STRING_LEN] = "";
test_req_t *request;
test_req_t params;
test_req_t ctx;
ctx.complete = 0;
params.op_attr_mask = UCP_OP_ATTR_FIELD_CALLBACK | UCP_OP_ATTR_FIELD_USER_DATA;
params.user_data = &ctx;
if (is_server) {
  while (!am_data_desc.complete) {
    ucp_worker_progress(ucp_worker);
  }
  am_data_desc.complete = 0;
  if (am_data_desc.is_rndv) {
    /* Rendezvous request has arrived, need to invoke receive operation to confirm data transfer from the sender to the "recv_message" buffer. */
    params.op_attr_mask = UCP_OP_ATTR_FLAG_NO_IMM_CMP1;
    params.cb.recv_am = am_recv_cb;
    request = ucp_am_recv_data_nbx(ucp_worker, am_data_desc.desc, &recv_message, TEST_STRING_LEN, &params);
  } else {
    /* Data has arrived eagerly and is ready for use, no need to initiate receive operation. */
    request = NULL;
  }
} else {
  /* Client sends a message to the server using the AM API */
  param.cb.send = (ucp_send_nbx_callback_t)send_cb,
  request = ucp_am_send_nbx(ep, TEST_AM_ID, NULL, 0ul, test_message, TEST_STRING_LEN, &params);
}
return request_finalize(ucp_worker, request, &ctx, is_server, recv_message, current_iter);
}
static void ep_close(ucp_worker_h ucp_worker, ucp_ep_h ep)
{
    ucp_request_param_t param;
    ucs_status_t status;
    void *close_req;
    param.op_attr_mask = UCP_OP_ATTR_FIELD_FLAGS;
    param.flags = UCP_EP_CLOSE_FLAG_FORCE;
    close_req = ucp_ep_close_nbx(ep, &param);
    if (UCS_PTR_IS_PTR(close_req)) {
        do {
            ucp_worker_progress(ucp_worker);
            status = ucp_request_check_status(close_req);
        } while (status == UCS_INPROGRESS);
        ucp_request_free(close_req);
    } else if (UCS_PTR_STATUS(close_req) != UCS_OK) {
        fprintf(stderr, "failed to close ep %p\n", (void*)ep);
    }
}
static void usage()
{
    fprintf(stderr, "Usage: ucp_client_server [parameters]\n";
    fprintf(stderr, "UCP client-server example utility\n";
    fprintf(stderr, "Parameters are:\n";
    fprintf(stderr, "  -a Set IP address of the server (required for client and should not be specified ";
    fprintf(stderr, "for the server)\n";
    fprintf(stderr, "  -l Set IP address where server listens (If not specified, server uses INADDR_ANY; ";
    fprintf(stderr, "irrelevant at client)\n";
    fprintf(stderr, "  -p Port number to listen/connect to (default = %d). ";
    fprintf(stderr, "0 on the server side means select a random port and print it\n", DEFAULT_PORT);
    fprintf(stderr, "  -c Communication type for the client and server. ";
    fprintf(stderr, "Valid values are:\n";
    fprintf(stderr, "    'stream': Stream API\n";
    fprintf(stderr, "    'tag': Tag API\n";
    fprintf(stderr, "    'am': AM API\n";
    fprintf(stderr, "If not specified, %s API will be used.\n", COMM_TYPE_DEFAULT);
    fprintf(stderr, "  -i Number of iterations to run. Client and server must ";
    fprintf(stderr, "have the same value. (default = %d).\n", num_iterations);
    fprintf(stderr, "\n";
    fprintf(stderr, "\n";
    fprintf(stderr, "\n";
    fprintf(stderr, "\n";
}
static int parse_cmd(int argc, char *const argv[], char **server_addr, char **listen_addr, send_recv_type_t *send_recv_type)
{
    int c = 0;
    int port;
    while ((c = getopt(argc, argv, "a:l:p:c:i:")) != -1) {
        switch (c) {
        case 'a':
            *server_addr = optarg;
            break;
        case 'l':
            *listen_addr = optarg;
            break;
        case 'p':
            port = atoi(optarg);
            if ((port < 0) || (port > UINT16_MAX)) {
                fprintf(stderr, "Wrong server port number %d\n", port);
                return -1;
            }
            server_port = port;
            break;
        case 'c':
            if (!strcasecmp(optarg, "stream")) {
                *send_recv_type = CLIENT_SERVER_SEND_RECV_STREAM;
            } else if (!strcasecmp(optarg, "tag")) {
                *send_recv_type = CLIENT_SERVER_SEND_RECV_TAG;
            } else if (!strcasecmp(optarg, "am")) {
                /* TODO: uncomment below when AM API is fully supported. */
                fprintf(stderr, "AM API is not fully supported yet\n");
                return -1;
            } else {
                fprintf(stderr, "Wrong communication type \%. ";
                fprintf(stderr, "Using \%s as default\n", optarg, COMM_TYPE_DEFAULT);
            }
            break;
        case 'i':
            num_iterations = atoi(optarg);
            break;
        }
    }
    return 0;
}
default:
    usage();
    return -1;
}

return 0;
}

static char* sockaddr_get_ip_str(const struct sockaddr_storage *sock_addr,
    char *ip_str, size_t max_size)
{
    struct sockaddr_in addr_in;
    struct sockaddr_in6 addr_in6;
    switch (sock_addr->ss_family) {
    case AF_INET:
        memcpy(&addr_in, sock_addr, sizeof(struct sockaddr_in));
        inet_ntop(AF_INET, &addr_in.sin_addr, ip_str, max_size);
        return ip_str;
    case AF_INET6:
        memcpy(&addr_in6, sock_addr, sizeof(struct sockaddr_in6));
        inet_ntop(AF_INET6, &addr_in6.sin6_addr, ip_str, max_size);
        return ip_str;
    default:
        return "Invalid address family";
    }
}

static char* sockaddr_get_port_str(const struct sockaddr_storage *sock_addr,
    char *port_str, size_t max_size)
{
    struct sockaddr_in addr_in;
    struct sockaddr_in6 addr_in6;
    switch (sock_addr->ss_family) {
    case AF_INET:
        memcpy(&addr_in, sock_addr, sizeof(struct sockaddr_in));
        snprintf(port_str, max_size, "%d", ntohs(addr_in.sin_port));
        return port_str;
    case AF_INET6:
        memcpy(&addr_in6, sock_addr, sizeof(struct sockaddr_in6));
        snprintf(port_str, max_size, "%d", ntohs(addr_in6.sin6_port));
        return port_str;
    default:
        return "Invalid address family";
    }
}

static int client_server_communication(ucp_worker_h worker, ucp_ep_h ep,
    send_recv_type_t send_recv_type,
    int is_server, int current_iter)
{
    int ret;
    switch (send_recv_type) {
    case CLIENT_SERVER_SEND_RECV_STREAM:
        /* Client-Server communication via Stream API */
        ret = send_recv_stream(worker, ep, is_server, current_iter);
        break;
    case CLIENT_SERVER_SEND_RECV_TAG:
        /* Client-Server communication via Tag-Matching API */
        ret = send_recv_tag(worker, ep, is_server, current_iter);
        break;
    case CLIENT_SERVER_SEND_RECV_AM:
        /* Client-Server communication via AM API. */
        ret = send_recv_am(worker, ep, is_server, current_iter);
        break;
    default:
        fprintf(stderr, "unknown send-recv type %d\n", send_recv_type);
        return -1;
    }
    return ret;
}

static int init_worker(ucp_context_h ucp_context, ucp_worker_h *ucp_worker)
{
    ucp_worker_params_t worker_params;
    ucs_status_t status;
    int ret = 0;
    memset(&worker_params, 0, sizeof(worker_params));
    worker_params.field_mask = UCP_WORKER_PARAM_FIELD_THREAD_MODE;
    worker_params.thread_mode = UCS_THREAD_MODE_SINGLE;
    status = ucp_worker_create(ucp_context, &worker_params, ucp_worker);
    if (status != UCS_OK) {
        fprintf(stderr, "failed to ucp_worker_create (ks)\n", ucs_status_string(status));
        ret = -1;
    }
    return ret;
}

static void server_conn_handle_cb(ucp_conn_request_h conn_request, void *arg)
{
    ucp_conn_ctxt_t *context = arg;
    ucp_conn_request_attr_t attr;
    char ip_str[IP_STRING_LEN];
char port_str[PORT_STRING_LEN];
ucs_status_t status;
attr.field_mask = UCP_CONN_REQUEST_ATTR_FIELD_CLIENT_ADDR;
status = ucp_conn_request_query(conn_request, &attr);
if (status == UCS_OK) {
    printf("Server received a connection request from client at address %s:%s\n", 
           sockaddr_get_ip_str(&attr.client_address, ip_str, sizeof(ip_str)),
           sockaddr_get_port_str(&attr.client_address, port_str, sizeof(port_str)));
    } else if (status != UCS_ERR_UNSUPPORTED) {
        fprintf(stderr, "failed to query the connection request (%s)\n", 
                ucs_status_string(status));
    } else {
        /* The server is already handling a connection request from a client, 
         * reject this new one */
        print("Rejecting a connection request. \
             Only one client at a time is supported.\n         ");
        status = ucp_listener_reject(context->listener, conn_request);
        if (status != UCS_OK) {
            fprintf(stderr, "server failed to reject a connection request: (%s)\n", 
                    ucs_status_string(status));
        }
    }
}

static ucs_status_t server_create_ep(ucp_worker_h data_worker,
                                   ucp_conn_request_h conn_request,
                                   ucp_ep_h *server_ep)
{
    ucp_ep_params_t ep_params;
    ucs_status_t status;
    /* Server creates an ep to the client on the data worker. \
     * This is not the worker the listener was created on. \
     * The client side should have initiated the connection, leading \
     * to this ep’s creation */
    ep_params.field_mask = UCP_EP_PARAM_FIELD_ERR_HANDLER | 
                           UCP_EP_PARAM_FIELD_CONN_REQUEST;
    ep_params.conn_request = conn_request;
    ep_params.err_handler.cb = err_cb;
    ep_params.err_handler.arg = NULL;
    status = ucp_ep_create(data_worker, &ep_params, server_ep);
    if (status != UCS_OK) {
        fprintf(stderr, "failed to create an endpoint on the server: (%s)\n", 
                ucs_status_string(status));
    }
    return status;
}

static ucs_status_t start_server(ucp_worker_h ucp_worker,
                                 ucx_server_ctx_t *context,
                                 ucp_listener_h *listener_p, const char *ip)
{
    struct sockaddr_in listen_addr;
    ucp_listener_params_t params;
    ucp_listener_attr_t attr;
    ucs_status_t status;
    char ip_str[IP_STRING_LEN];
    char port_str[PORT_STRING_LEN];
    set_listen_addr(ip, &listen_addr);
    params.field_mask = UCP_LISTENER_PARAM_FIELD_SOCK_ADDR | 
                        UCP_LISTENER_PARAM_FIELD_CONN_HANDLER;
    params.sockaddr.addr = (const struct sockaddr *)&listen_addr;
    params.sockaddr.addrlen = sizeof(listen_addr);
    params.conn_handler.cb = server_conn_handle_cb;
    params.conn_handler.arg = context;
    /* Create a listener on the server side to listen on the given address. */
    status = ucp_listener_create(ucp_worker, &params, listener_p);
    if (status != UCS_OK) {
        fprintf(stderr, "failed to listen (%s)\n", ucs_status_string(status));
        goto out;
    }
    /* Query the created listener to get the port it is listening on. */
    attr.field_mask = UCP_LISTENER_ATTR_FIELD_SOCK_ADDR;
    status = ucp_listener_query(listener_p, &attr);
    if (status != UCS_OK) {
        fprintf(stderr, "failed to query the listener (%s)\n", 
                ucs_status_string(status));
        ucp_listener_destroy(listener_p);
        goto out;
    }
    fprintf(stderr, "server is listening on IP %s port %s\n", 
            sockaddr_get_ip_str(&attr.sockaddr, ip_str, IP_STRING_LEN),
            sockaddr_get_port_str(&attr.sockaddr, port_str, PORT_STRING_LEN));
    printf("Waiting for connection...\n");
out: 
    return status;
}
static int client_server_do_work(ucp_worker_h ucp_worker, ucp_ep_h ep, send_recv_type_t send_recv_type, int is_server)
{
    int i, ret = 0;
    for (i = 0; i < num_iterations; i++) {
        ret = client_server_communication(ucp_worker, ep, send_recv_type, is_server, i);
        if (ret != 0) {
            fprintf(stderr, "%s failed on iteration #%d
", (is_server ? "server" : "client"), i + 1);
            goto out;
        }
    }
out:
    return ret;
}

static int run_server(ucp_context_h ucp_context, ucp_worker_h ucp_worker, char *listen_addr, send_recv_type_t send_recv_type)
{
    ucx_server_ctx_t context;
    ucp_worker_h ucp_data_worker;
    ucp_am_handler_param_t param;
    ucp_ep_h server_ep;
    ucs_status_t status;
    int ret;
    /* Create a data worker (to be used for data exchange between the server
    * and the client after the connection between them was established) */
    ret = init_worker(ucp_context, &ucp_data_worker);
    if (ret != 0) {
        goto err;
    }
    if (send_recv_type == CLIENT_SERVER_SEND_RECV_AM) {
        /* Initialize Active Message data handler */
        param.field_mask = UCP_AM_HANDLER_PARAM_FIELD_ID |
            UCP_AM_HANDLER_PARAM_FIELD_CB |
            UCP_AM_HANDLER_PARAM_FIELD_ARG;
        param.id = TEST_AM_ID;
        param.cb = ucp_am_data_cb;
        param.arg = ucp_data_worker; /* not used in our callback */
        status = ucp_worker_set_am_recv_handler(ucp_data_worker, &param);
        if (status != UCS_OK) {
            ret = -1;
            goto err_worker;
        }
    }
    /* Initialize the server’s context. */
    context.conn_request = NULL;
    /* Create a listener on the worker created at first. The ‘connection
    * worker’ - used for connection establishment between client and server.
    * This listener will stay open for listening to incoming connection
    * requests from the client */
    status = start_server(ucp_worker, &context, &context.listener, listen_addr);
    if (status != UCS_OK) {
        ret = -1;
        goto err_worker;
    }
    /* Server is always up listening */
    while (1) {
        /* Wait for the server to receive a connection request from the client.
        * If there are multiple clients for which the server’s connection request
        * callback is invoked, i.e. several clients are trying to connect in
        * parallel, the server will handle only the first one and reject the rest */
        while (context.conn_request != NULL) {
            ucp_worker_progress(ucp_worker);
        }
        /* Server creates an ep to the client on the data worker. 
        * This is not the worker the listener was created on. 
        * The client side should have initiated the connection, leading 
        * to this ep’s creation */
        status = server_create_ep(ucp_data_worker, context.conn_request, &server_ep);
        if (status != UCS_OK) {
            ret = -1;
            goto err_listener;
        }
        /* The server waits for all the iterations to complete before moving on 
        * to the next client */
        ret = client_server_do_work(ucp_data_worker, server_ep, send_recv_type, 1);
        if (ret != 0) {
            goto err_ep;
        }
        /* Close the endpoint to the client */
        ep_close(ucp_data_worker, server_ep);
        /* Reinitialize the server’s context to be used for the next client */
        context.conn_request = NULL;
    }
}

© 2021 Unified Communication X (UCX). All rights reserved.
printf("Waiting for connection...
");
}
err_ep:
ep_close(ucp_data_worker, server_ep);
err_listener:
ucp_listener_destroy(context.listener);
err_worker:
ucp_worker_destroy(ucp_data_worker);
err:
return ret;
}
static int run_client(ucp_worker_h ucp_worker, char *server_addr, send_recv_type_t send_recv_type)
{
ucp_ep_h client_ep;
ucs_status_t status;
int ret;
status = start_client(ucp_worker, server_addr, &client_ep);
if (status != UCS_OK) {
    fprintf(stderr, "failed to start client \(\%s\)\n", ucs_status_string(status));
    ret = -1;
    goto out;
}
ret = client_server_do_work(ucp_worker, client_ep, send_recv_type, 0);
/* Close the endpoint to the server */
ep_close(ucp_worker, client_ep);
out:
return ret;
}
static int init_context(ucp_context_h *ucp_context, ucp_worker_h *ucp_worker, send_recv_type_t send_recv_type)
{
    /* UCP objects */
    ucp_params_t ucp_params;
    ucs_status_t status;
    int ret = 0;
    memset(&ucp_params, 0, sizeof(ucp_params));
    /* UCP initialization */
    ucp_params.field_mask = UCP_PARAM_FIELD_FEATURES;
    if (send_recv_type == CLIENT_SERVER_SEND_RECV_STREAM) {
        ucp_params.features = UCP_FEATURE_STREAM;
    } else if (send_recv_type == CLIENT_SERVER_SEND_RECV_TAG) {
        ucp_params.features = UCP_FEATURE_TAG;
    } else {
        ucp_params.features = UCP_FEATURE_AM;
    }
    status = ucp_init(&ucp_params, NULL, ucp_context);
    if (status != UCS_OK) {
        fprintf(stderr, "failed to ucp_init \(\%s\)\n", ucs_status_string(status));
        ret = -1;
        goto err;
    }
    ret = init_worker(*ucp_context, ucp_worker);
    if (ret != 0) {
        goto err_cleanup;
    }
    return ret;
}
err_cleanup:
ucp_cleanup(*ucp_context);
err:
return ret;
}
int main(int argc, char **argv)
{
    send_recv_type_t send_recv_type = CLIENT_SERVER_SEND_RECV_DEFAULT;
    char *server_addr = NULL;
    char *listen_addr = NULL;
    int ret;
    /* UCP objects */
    ucp_context_h ucp_context;
    ucp_worker_h ucp_worker;
    ret = parse_cmd(argc, argv, &server_addr, &listen_addr, &send_recv_type);
    if (ret != 0) {
        goto err;
    }
    /* Initialize the UCX required objects */
    ret = init_context(&ucp_context, &ucp_worker, send_recv_type);
    if (ret != 0) {
        goto err;
    }
    /* Client-Server initialization */
    if (server_addr == NULL) {
        /* Server side */
        ret = run_server(ucp_context, ucp_worker, listen_addr, send_recv_type);
    } else {
        /* Client side */
        ret = run_client(ucp_worker, server_addr, send_recv_type);
    }
    return ret;
}
ucp_worker_destroy(ucp_worker);
ucp_cleanup(ucp_context);
err:
    return ret;
}

8.2 ucp_hello_world.c

UCP hello world client / server example utility.

UCP hello world client / server example utility.

Server side:

Client side:

Client acquires Server UCX address via TCP socket

Author:

Ilya Nelkenbaum <ilya@nelkenbaum.com>
Sergey Shalnov <sergeysh@mellanox.com> 7-June-2016

1. Include "hello_world_util.h"
2. Include <ucp/api/ucp.h>
3. Include <sys/socket.h>
4. Include <sys/types.h>
5. Include <sys/epoll.h>
6. Include <netinet/in.h>
7. Include <assert.h>
8. Include <netdb.h>
9. Include <stdio.h>
10. Include <stdlib.h>
11. Include <string.h>
12. Include <unistd.h> / * getopt */
13. Include <ctype.h> / * isprint */
14. Include <pthread.h> / * pthread_self */
15. Include <errno.h> / * errno */
16. Include <time.h>
17. Include <signal.h> / * raise */
18. struct msg {
    uint64_t data_len;
};
19. struct ucx_context {
    int completed;
};
20. enum ucp_test_mode_t {
    TEST_MODE_PROBE,
    TEST_MODE_WAIT,
    TEST_MODE_EVENTFD
} ucp_test_mode = TEST_MODE_PROBE;
21. typedef enum {
    FAILURE_MODE_NONE,
    FAILURE_MODE_SEND, /* fail send operation on server */
    FAILURE_MODE_RECV, /* fail receive operation on client */
    FAILURE_MODE_REPFAILIVE /* fail without communication on client */
} failure_mode_t;
22. static struct err_handling {
    ucp_err_handling_mode_t ucp_err_mode;
    failure_mode_t failure_mode;
} err_handling_opt;
23. static ucs_status_t ep_status = UCS_OK;
24. static uint16_t server_port = 13337;
25. static long test_string_length = 16;
26. static const ucp_tag_t tag = 0x1337a880u;
27. static const ucp_tag_t tag_mask = UINT64_MAX;
28. static const char *addr_msg_str = "UCX address message";
29. static const char *data_msg_str = "UCX data message";
30. static ucp_address_t *local_addr;
31. static ucp_address_t *peer_addr;
static size_t local_addr_len;
static size_t peer_addr_len;
static ucs_status_t parse_cmd(int argc, char * const argv[], char **server_name);
static void set_msg_data_len(struct msg *msg, uint64_t data_len);

static void request_init(void *request)
{
    struct ucx_context *context = (struct ucx_context *)request;
    context->completed = 0;
}

static void send_handler(void *request, ucs_status_t status, void *ctx)
{
    struct ucx_context *context = (struct ucx_context *)request;
    const char *str = (const char *)ctx;
    context->completed = 1;
    printf("[0x%x] send handler called for \%s with status %d (%s)\n",
           (unsigned int)pthread_self(), str, status,
           ucs_status_string(status));
}

static void failure_handler(void *arg, ucp_ep_h ep, ucs_status_t status)
{
    ucs_status_t *arg_status = (ucs_status_t *)arg;
    printf("[0x%x] failure handler called with status %d (%s)\n",
           (unsigned int)pthread_self(), status, ucs_status_string(status));
    *arg_status = status;
}

static void recv_handler(void *request, ucs_status_t status,
                         ucp_tag_recv_info_t *info)
{
    struct ucx_context *context = (struct ucx_context *)request;
    context->completed = 1;
    printf("[0x%x] receive handler called with status %d (%s), length %lu\n",
           (unsigned int)pthread_self(), status, ucs_status_string(status),
           info->length);
}

static ucs_status_t ucx_wait(ucp_worker_h ucp_worker, struct ucx_context *request,
                             const char *op_str, const char *data_str)
{
    ucs_status_t status;
    if (UCS_PTR_IS_ERR(request)) {
        status = UCS_PTR_STATUS(request);
    } else if (UCS_PTR_IS_PTR(request)) {
        while (!request->completed) {
            ucp_worker_progress(ucp_worker);
        }
        request->completed = 0;
        status = ucp_request_check_status(request);
        ucp_request_release(request);
    } else {
        status = UCS_OK;
    }
    if (status != UCS_OK) {
        fprintf(stderr, "unable to %s %s (%s)\n", op_str, data_str,
                ucs_status_string(status));
    } else {
        printf("finish to %s %s\n", op_str, data_str);
    }
    return status;
}

static ucs_status_t test_poll_wait(ucp_worker_h ucp_worker)
{
    int err = 0;
    ucs_status_t ret = UCS_ERR_NO_MESSAGE;
    int epoll_fd_local = 0;
    int epoll_fd = 0;
    ucs_status_t status;
    struct epoll_event ev;
    ev.data.u64 = 0;
    status = ucp_worker_get_efd(ucp_worker, &epoll_fd);
    CHKERR_JUMP(UCS_OK != status, "ucp_worker_get_efd", err);
    /\ It is recommended to copy original fd */
    epoll_fd_local = epoll_create(1);
    ev.data.fd = epoll_fd;
    ev.events = EPOLLIN;
    err = epoll_ctl(epoll_fd_local, EPOLL_CTL_ADD, epoll_fd, &ev);
    CHKERR_JUMP(err < 0, "add original socket to the new epoll", err);
    /\ Need to prepare ucp_worker before epoll_wait */
    status = ucp_worker_arm(ucp_worker);
    if (status != UCS_ERR_BUSY) { /* some events are arrived already */
        ret = UCS_OK;
        goto err;
    }
    CHKERR_JUMP(status != UCS_OK, "ucp_worker_arm\n", err);
    do {
        err = epoll_wait(epoll_fd_local, &ev, 1, -1);
    } while (err > 0);
while ((err == -1) && (errno == EINTR)) {
    ret = UCS_OK;
    err_fd:
    close(epoll_fd_local);
    err:
    return ret;
}

static int run_ucx_client(ucp_worker_h ucp_worker) {
    struct msg *msg = NULL;
    size_t msg_len = 0;
    int ret = -1;
    ucp_request_param_t send_param;
    ucp_tag_recv_info_t info_tag;
    ucp_tag_message_h msg_tag;
    ucs_status_t status;
    ucp_ep_h server_ep;
    ucp_ep_params_t ep_params;
    struct ucx_context *request;
    char *str;
    /* Send client UCX address to server */
    ep_params.field_mask = UCP_EP_PARAM_FIELD_REMOTE_ADDRESS |
                           UCP_EP_PARAM_FIELD_ERR_HANDLING_MODE |
                           UCP_EP_PARAM_FIELD_ERR_HANDLER |
                           UCP_EP_PARAM_FIELD_USER_DATA;
    ep_params.address = peer_addr;
    ep_params.err_mode = err_handling_opt.ucp_err_mode;
    ep_params.err_handler.cb = failure_handler;
    ep_params.err_handler.arg = NULL;
    ep_params.user_data = &ep_status;
    status = ucp_ep_create(ucp_worker, &ep_params, &server_ep);
    CHKERR_JUMP(status != UCS_OK, "ucp_ep_create
    ", err);
    msg_len = sizeof(*msg) + local_addr_len;
    msg = malloc(msg_len);
    CHKERR_JUMP(msg == NULL, "allocate memory
    ", err_ep);
    memset(msg, 0, msg_len);
    msg->data_len = local_addr_len;
    memcpy(msg + 1, local_addr, local_addr_len);
    send_param.op_attr_mask = UCP_OP_ATTR_FIELD_CALLBACK |
                            UCP_OP_ATTR_FIELD_USER_DATA;
    send_param.cb.send = send_handler;
    send_param.user_data = (void *)addr_msg_str;
    request = ucp_tag_send_nbx(server_ep, msg, msg_len, tag,
                                &send_param);
    status = ucx_wait(ucp_worker, request, "send",
                      addr_msg_str);
    if (status != UCS_OK) {
        free(msg);
        goto err_ep;
    }
    free(msg);
    if (err_handling_opt.failure_mode == FAILURE_MODE_RECV) {
        fprintf(stderr, "Emulating failure before receive operation on client side\n");
        raise(SIGKILL);
    }
    /* Receive test string from server */
    for (;;) {
        CHKERR_JUMP(ep_status != UCS_OK, "receive data: EP disconnected\n", err_ep);
        /* Probing incoming events in non-block mode */
        msg_tag = ucp_tag_probe_nb(ucp_worker, tag, tag_mask, 1, &info_tag);
        if (msg_tag != NULL) { /* Message arrived */
            break;
        } else if (ucp_worker_progress(ucp_worker)) {
            /* Some events were polled; try again without going to sleep */
            continue;
        } /* If we got here, ucp_worker_progress() returned 0, so we can sleep.
        * Following blocked methods used to polling internal file descriptor
        * to make CPU idle and don’t spin loop*/
        if (ucp_test_mode == TEST_MODE_WAIT) {
            /* Polling incoming events*/
            status = ucp_worker_wait(ucp_worker);
            CHKERR_JUMP(status != UCS_OK, "ucp_worker_wait\n", err_ep);
        } else if (ucp_test_mode == TEST_MODE_EVENTFD) {
            status = test_poll_wait(ucp_worker);
            CHKERR_JUMP(status != UCS_OK, "test_poll_wait\n", err_ep);
        }
    }
    if (err_handling_opt.failure_mode == FAILURE_MODE_KEEPALIVE) {
        fprintf(stderr, "Emulating unexpected failure after receive completion
    * on client side, server should detect error by *
    * keepalive mechanism\n")
        raise(SIGKILL);
    }
}
msg = mem_type_malloc(info_tag.length);
CHKERR_JUMP(msg == NULL, "allocate memory\n", err_ep);
request = ucp_tag_msg_recv_nb(ucp_worker, msg, info_tag.length,
ucp_dt_make_contig(1), msg_tag, recv_handler);
status = ucx_wait(ucp_worker, request, "receive", data_msg_str);
if (status != UCS_OK) {
    mem_type_free(msg);
    goto err_ep;
}
str = calloc(1, test_string_length);
if (str != NULL) {
    mem_type_memcpy(str, msg + 1, test_string_length);
    printf("\n--- UCP TEST SUCCESS ---\n\n");
    printf("%s", str);
    printf("\n------------------\n\n");
    free(str);
} else {
    fprintf(stderr, "Memory allocation failed\n");
    mem_type_free(msg);
    goto err_ep;
}
mem_type_free(msg);
ret = 0;
err_ep:
ucp_ep_destroy(server_ep);
err:
return ret;
}
static void flush_callback(void *request, ucs_status_t status, void *user_data)
{
}
static ucs_status_t flush_ep(ucp_worker_h worker, ucp_ep_h ep)
{
    ucp_request_param_t param;
    void *request;
    param.op_attr_mask = UCP_OP_ATTR_FIELD_CALLBACK;
    param.cb.send = flush_callback;
    request = ucp_ep_flush_nbx(ep, &param);
    if (request == NULL) {
        return UCS_OK;
    } else if (UCS_PTR_IS_ERR(request)) {
        return UCS_PTR_STATUS(request);
    } else {
        ucs_status_t status;
        do {
            ucp_worker_progress(worker);
            status = ucp_request_check_status(request);
        } while (status == UCS_INPROGRESS);
        ucp_request_release(request);
        return status;
    }
}
static int run_ucx_server(ucp_worker_h ucp_worker)
{
    struct msg *msg = NULL;
    struct ucx_context *request = NULL;
    size_t msg_len = 0;
    ucp_request_param_t send_param;
    ucp_tag_recv_info_t info_tag;
    ucp_tag_message_h msg_tag;
    ucs_status_t status;
    ucp_ep_h client_ep;
    ucp_ep_params_t ep_params;
    int ret;
    /* Receive client UCX address */
    do {
        /* Progressing before probe to update the state */
        ucp_worker_progress(ucp_worker);
        /* Probing incoming events in non-block mode */
        msg_tag = ucp_tag_probe_nb(ucp_worker, tag, tag_mask, 1, &info_tag);
    } while (msg_tag == NULL);
    msg = malloc(info_tag.length);
    CHKERR_ACTION(msg == NULL, "allocate memory\n", ret = -1; goto err);
    request = ucp_tag_msg_recv_nb(ucp_worker, msg, info_tag.length,
ucp_dt_make_contig(1), msg_tag, recv_handler);
    status = ucx_wait(ucp_worker, request, "receive", addr_msg_str);
    if (status != UCS_OK) {
        free(msg);
        ret = -1;
        goto err;
    }
    if (err_handling_opt.failure_mode == FAILURE_MODE_SEND) {
        fprintf(stderr, "Emulating unexpected failure on server side, client "
            "should detect error by keepalive mechanism\n");
        free(msg);
        raise(SIGKILL);
    }
peer_addr_len = msg->data_len;
peer_addr = malloc(peer_addr_len);
if (peer_addr == NULL) {
    fprintf(stderr, "unable to allocate memory for peer address\n");
    free(msg);
    ret = -1;
    goto err;
}
memcpy(peer_addr, msg + 1, peer_addr_len);
free(msg);
/* Send test string to client */
ep_params.field_mask = UCP_EP_PARAM_FIELD_REMOTE_ADDRESS |
    UCP_EP_PARAM_FIELD_ERR_HANDLING_MODE |
    UCP_EP_PARAM_FIELD_ERR_HANDLER |
    UCP_EP_PARAM_FIELD_USER_DATA;
ep_params.address = peer_addr;
ep_params.err_mode = err_handling_opt.ucp_err_mode;
ep_params.err_handler.cb = failure_handler;
ep_params.err_handler.arg = NULL;
ep_params.user_data = &ep_status;
status = ucp_ep_create(ucp_worker, &ep_params, &client_ep);
/* If peer failure testing was requested, it could be possible that UCP EP
* couldn't be created; in this case set 'ret = 0' to report success */
ret = (err_handling_opt.failure_mode != FAILURE_MODE_NONE) ? 0 : -1;
CHKERRQ_ACTION(status != UCS_OK, "ucp_ep_create
    
    n", goto err);
msg_len = sizeof(*msg) + test_string_length;
msg = mem_type_malloc(msg_len);
CHKERRQ_ACTION(msg == NULL, "allocate memory
    
    n", ret = -1; goto err_ep);
mem_type_memset(msg, 0, msg_len);
set_msg_data_len(msg, msg_len - sizeof(*msg));
ret = generate_test_string((char *)(msg + 1), test_string_length);
CHKERRQ_JUMP(ret < 0, "generate test string", err_free_mem_type_msg);
if (err_handling_opt.failure_mode == FAILURE_MODE_RECV) {
    /* Sleep for small amount of time to ensure that client was killed
    * and peer failure handling is covered */
    sleep(5);
}
ucp_worker_progress(ucp_worker);
send_param.op_attr_mask = UCP_OP_ATTR_FIELD_CALLBACK |
    UCP_OP_ATTR_FIELD_USER_DATA |
    UCP_OP_ATTR_FIELD_MEMORY_TYPE;
send_param.cb.send = send_handler;
send_param.user_data = (void *)data_msg_str;
send_param.memory_type = test_mem_type;
request = ucp_tag_send_nbx(client_ep, msg, msg_len, tag,
    &send_param);
status = ucx_wait(ucp_worker, request, "send",
    data_msg_str);
if (status != UCS_OK) {
    if (err_handling_opt.failure_mode != FAILURE_MODE_NONE) {
        ret = -1;
    } else {
        /* If peer failure testing was requested, set 'ret = 0' to report
        * success from the application */
        ret = 0;
        /* Make sure that failure_handler was called */
        while (ep_status == UCS_OK) {
            ucp_worker_progress(ucp_worker);
        }
    }
    goto err_free_mem_type_msg;
}
if (err_handling_opt.failure_mode == FAILURE_MODE_KEEPALIVE) {
    fprintf(stderr, "Waiting for client is terminated\n");
    while (ep_status == UCS_OK) {
        ucp_worker_progress(ucp_worker);
    }
    status = flush_ep(ucp_worker, client_ep);
    printf("flush_ep completed with status %d (\$s)\n",
        status, ucs_status_string(status));
    ret = 0;
    err_free_mem_type_msg;
    mem_type_free(msg);
    err_ep:
    ucp_ep_destroy(client_ep);
    err:
    return ret;
}
static int run_test(const char *client_target_name, ucp_worker_h ucp_worker)
{
    if (client_target_name != NULL) {
        return run_ucx_client(ucp_worker);
    } else {
        return run_ucx_server(ucp_worker);
    }
int main(int argc, char **argv) {
    int oob_sock = -1;
    memset(&ucp_params, 0, sizeof(ucp_params));
    memset(&worker_params, 0, sizeof(worker_params));

    if (argc < 2) {
        printf("Usage: %s <client_target_name>
", argv[0]);
        return -1;
    }

    /* Parse the command line */
    client_target_name = argv[1];

    /* UCP initialization */
    status = ucp_init(&ucp_params, config, &ucp_context);
    CHKERR_JUMP(status != UCS_OK, "ucp_init
", err);

    worker_params.field_mask = UCP_WORKER_PARAM_FIELD_THREAD_MODE;
    worker_params.thread_mode = UCS_THREAD_MODE_SINGLE;
    status = ucp_worker_create(ucp_context, &worker_params, &ucp_worker);
    CHKERR_JUMP(status != UCS_OK, "ucp_worker_create
", err_cleanup);

    if (client_target_name) {
        peer_addr_len = local_addr_len;
        oob_sock = client_connect(client_target_name, server_port);
        CHKERR_JUMP(oob_sock < 0, "client_connect
", err);
        ret = recv(socket, peer_addr, peer_addr_len, MSG_WAITALL);
        CHKERR_JUMP_RETVAL(ret != (int)peer_addr_len,
                           "recv address
", err_peer_addr, ret);
        peer_addr = malloc(peer_addr_len);
        CHKERR_JUMP(!peer_addr, "allocate memory
", err);
        ret = send(socket, peer_addr, peer_addr_len, MSG_WAITALL);
        CHKERR_JUMP_RETVAL(ret != (int)peer_addr_len,
                           "send address
", err_peer_addr, ret);
    } else {

    }

    return ret;
}

err_peer_addr:
    free(peer_addr);
    ucp_worker_release_address(ucp_worker, local_addr);
err_worker:
    ucp_worker_destroy(ucp_worker);
err_cleanup:
    ucp_cleanup(ucp_context);
err:
    return ret;
}
static void print_usage()
{
    fprintf(stderr, "Usage: ucp_hello_world [parameters]\n");
    fprintf(stderr, "UCP hello world client/server example utility\n";
    fprintf(stderr, "$n";
    fprintf(stderr, "Select test mode \"wait\" to test \nucp_worker_wait function\n";
    fprintf(stderr, "$n";
    fprintf(stderr, "$n";
    fprintf(stderr, \"error with enabled UCP_ERR_HANDLING_MODE_PEER\n\";
    fprintf(stderr, \"send - send failure on server side \nbefore send initiated\n\";
    fprintf(stderr, \"recv - receive failure on client side \nbefore receive completed\n\";
    fprintf(stderr, \"keepalive - keepalive failure on client side \nafter communication completed\n\";
    print_common_help();
    fprintf(stderr, "$n";
}

ucs_status_t parse_cmd(int argc, char * const argv[], char **server_name)
{
    int c = 0, idx = 0;
    opterr = 0;
    err_handling_opt.ucp_err_mode = UCP_ERR_HANDLING_MODE_NONE;
    err_handling_opt.failure_mode = FAILURE_MODE_NONE;
    while ((c = getopt(argc, argv, "wfbe:n:p:s:m:h")) != -1) {
        switch (c) { 
        case 'w':
            ucp_test_mode = TEST_MODE_WAIT;
            break;
        case 'f':
            ucp_test_mode = TEST_MODE_EVENTFD;
            break;
        case 'b':
            ucp_test_mode = TEST_MODE_PROBE;
            break;
        case 'e':
            err_handling_opt.ucp_err_mode = UCP_ERR_HANDLING_MODE_PEER;
            if (!strcmp(optarg, "recv")) {
                err_handling_opt.failure_mode = FAILURE_MODE_RECV;
            } else if (!strcmp(optarg, "send")) {
                err_handling_opt.failure_mode = FAILURE_MODE_SEND;
            } else if (!strcmp(optarg, "keepalive")) {
                err_handling_opt.failure_mode = FAILURE_MODE_KEEPALIVE;
            } else {
                print_usage();
                return UCS_ERR_UNSUPPORTED;
            }
            break;
        case 'n':
            *server_name = optarg;
            break;
        case 'p':
            server_port = atoi(optarg);
            if (server_port <= 0) {
                fprintf(stderr, "Wrong server port number %d\n", server_port);
                return UCS_ERR_UNSUPPORTED;
            }
            break;
        case 's':
            test_string_length = atol(optarg);
            if (test_string_length <= 0) {
                fprintf(stderr, "Wrong string size %ld\n", test_string_length);
                return UCS_ERR_UNSUPPORTED;
            }
            break;
        case 'm':
            test_mem_type = parse_mem_type(optarg);
            if (test_mem_type == UCS_MEMORY_TYPE_LAST) {
                return UCS_ERR_UNSUPPORTED;
            }
            break;
        case '?':
            if (optopt == 's') {
                fprintf(stderr, "Option -c requires an argument.\n", optopt);
            } else if (isprint (optopt)) {
                fprintf(stderr, "Unknown option '-%c'.\n", optopt);
            } else {
                fprintf(stderr, "Unknown option character '\%c'.\n", optopt);
            }
            // Fall through ./
        case 'h':
            default:
                print_usage();
                break;
        }
8.3 uct_hello_world.c

UCT hello world client / server example utility.

```c
#include "hello_world_util.h"
#include <limits.h>
#include <uct/api/uct.h>
#include <assert.h>
#include <ctype.h>

typedef enum {
    FUNC_AM_SHORT,
    FUNC_AM_BCOPY,
    FUNC_AM_ZCOPY
} func_am_t;

typedef struct {
    int is_uct_desc;
} recv_desc_t;

typedef struct {
    char *server_name;
    uint16_t server_port;
    func_am_t func_am_type;
    const char *dev_name;
    const char *tl_name;
    long test_strlen;
} cmd_args_t;

typedef struct {
    uct_iface_attr_t iface_attr; /* Interface attributes: capabilities and limitations */
    uct_iface_h iface; /* Communication interface context */
    uct_md_attr_t md_attr; /* Memory domain attributes: capabilities and limitations */
    uct_md_h md; /* Memory domain */
    uct_worker_h worker; /* Workers represent allocated resources in a communication thread */
} iface_info_t;

/* Helper data type for am_short */
typedef struct {
    uint64_t header;
    char *payload;
    size_t len;
} someone_t;

/* Helper data type for am_bcopy */
typedef struct {
    char *data;
    size_t len;
} someone_t;

/* Helper data type for am_zcopy */
typedef struct {
    uct_completion_t uct_comp;
    uct_md_h md;
    uct_mem_h memh;
} someone_t;

static void *desc_holder = NULL;
static size_t func_am_max_size(func_am_t func_am_type, const uct_iface_attr_t *attr)
{ switch (func_am_type) {
    case FUNC_AM_SHORT:
        return "uct_ep_am_short";
    case FUNC_AM_BCOPY:
        return "uct_ep_am_bcopy";
    case FUNC_AM_ZCOPY:
        return "uct_ep_am_zcopy";
    }
    return NULL;
}

static size_t func_am_max_size(func_am_t func_am_type,
    const uct_iface_attr_t *attr)
{ switch (func_am_type) {
    case FUNC_AM_SHORT:
        return attr->cap.am.max_short;
    case FUNC_AM_BCOPY:
        return attr->cap.am.max_bcopy;
    case FUNC_AM_ZCOPY:
        return attr->cap.am.max_zcopy;
    }
}
```

© 2021 Unified Communication X (UCX). All rights reserved.
Example Documentation

```c
return 0;

/* Helper function for am_short */
void am_short_params_pack(char *buf, size_t len, am_short_args_t *args)
{
    args->header = *(uint64_t *)buf;
    if (len > sizeof(args->header)) {
        args->payload = (buf + sizeof(args->header));
        args->len = len - sizeof(args->header);
    } else {
        args->payload = NULL;
        args->len = 0;
    }
}

ucs_status_t do_am_short(iface_info_t *if_info, uct_ep_h ep, uint8_t id,
                         const cmd_args_t *cmd_args, char *buf)
{
    ucs_status_t status;
    am_short_args_t send_args;
    am_short_params_pack(buf, cmd_args->test_strlen, &send_args);
    do {
        /* Send active message to remote endpoint */
        status = uct_ep_am_short(ep, id, send_args.header, send_args.payload,
                                 send_args.len);
        uct_worker_progress(if_info->worker);
    } while (status == UCS_ERR_NO_RESOURCE);
    return status;
}

/* Pack callback for am_bcopy */
size_t am_bcopy_data_pack_cb(void *dest, void *arg)
{
    am_bcopy_args_t *bc_args = arg;
    mem_type_memcpy(dest, bc_args->data, bc_args->len);
    return bc_args->len;
}

ucs_status_t do_am_bcopy(iface_info_t *if_info, uct_ep_h ep, uint8_t id,
                         const cmd_args_t *cmd_args, char *buf)
{
    am_bcopy_args_t args;
    ssize_t len;
    args.data = buf;
    args.len = cmd_args->test_strlen;
    /* Send active message to remote endpoint */
    do {
        len = uct_ep_am_bcopy(ep, id, am_bcopy_data_pack_cb, &args, 0);
        uct_worker_progress(if_info->worker);
    } while (len == UCS_ERR_NO_RESOURCE);
    /* Negative len is an error code */
    return (len >= 0) ? UCS_OK : (ucs_status_t)len;
}

/* Completion callback for am_zcopy */
void zcopy_completion_cb(uct_completion_t *self)
{
    zcopy_comp_t *comp = (zcopy_comp_t *)self;
    assert((comp->uct_comp.count == 0) && (self->status == UCS_OK));
    if (comp->memh != UCT_MEM_HANDLE_NULL) {
        uct_md_mem_dereg(comp->md, comp->memh);
    }
    desc_holder = (void *)0xDEADBEEF;
}

ucs_status_t do_am_zcopy(iface_info_t *if_info, uct_ep_h ep, uint8_t id,
                         const cmd_args_t *cmd_args, char *buf)
{
    ucs_status_t status = UCS_OK;
    uct_mem_h memh;
    uct iov_t iov;
    zcopy_comp_t comp;
    if (if_info->md_attr.cap.flags & UCT_MD_FLAG_NEED_MEMH) {
        status = uct_md_mem_reg(if_info->md, buf, cmd_args->test_strlen,
                                 UCT_MD_MEM_ACCESS_RMA, &memh);
    } else {
        memh = UCT_MEM_HANDLE_NULL;
    }
    iov.buffer = buf;
    iov.length = cmd_args->test_strlen;
    iov.memh = memh;
    iov.stride = 0;
    iov.count = 1;
    comp.uct_comp.func = zcopy_completion_cb;
    comp.uct_comp.count = 1;
    comp.uct_comp.status = UCS_OK;
    comp.md = if_info->md;
    comp.memh = memh;
    if (status == UCS_OK) {
        do {
            status = uct_ep_am_zcopy(ep, id, NULL, 0, &iov, 1, 0,
```
(uct_completion_t *)&comp);
    uct_worker_progress(if_info->worker);
} while (status == UCS_ERR_NO_RESOURCE);
if (status == UCS_INPROGRESS) {
    while (!desc_holder) {
        /* Explicitly progress outstanding active message request */
        uct_worker_progress(if_info->worker);
    }
    status = UCS_OK;
}
}
return status;
}
static void print_strings(const char *label, const char *local_str,
const char *remote_str, size_t length)
{
    fprintf(stdout, "--- UCT TEST SUCCESS ---

["%s"] %s sent %s", label, local_str, remote_str);
    fprintf(stdout, "----------\n    fflush(stdout);
}
/* Callback to handle receive active message */
static ucs_status_t hello_world(void *arg, void *data, size_t length,
unsigned flags)
{
    func_am_t func_am_type = *(func_am_t *)arg;
    recv_desc_t *rdesc;
    print_strings("callback", func_am_t_str(func_am_type), data, length);
    if (flags & UCT_CB_PARAM_FLAG_DESC) {
        rdesc = (recv_desc_t *)data - 1;
        /* Hold descriptor to release later and return UCS_INPROGRESS */
        rdesc->is_uct_desc = 1;
        desc_holder = rdesc;
        return UCS_INPROGRESS;
    }
    /* We need to copy-out data and return UCS_OK if want to use the data
     * outside the callback */
    rdesc = malloc(sizeof(*rdesc) + length);
    CHKERR_ACTION(rdesc == NULL, "allocate memory

    memcopy(rdesc + 1, data, length);
    desc_holder = rdesc;
    return UCS_OK;
}
/* Init the transport by its name */
static ucs_status_t init_iface(char *dev_name, char *tl_name,
func_am_t func_am_type,
iface_info_t *iface_p)
{
    ucs_status_t status;
    uct_iface_config_t *config; /* Defines interface configuration options */
    uct_iface_params_t params;
    params.field_mask = UCT_IFACE_PARAM_FIELD_OPEN_MODE |
UCT_IFACE_PARAM_FIELD_DEVICE |
UCT_IFACE_PARAM_FIELD_STATS_ROOT |
UCT_IFACE_PARAM_FIELD_RX_HEADROOM |
UCT_IFACE_PARAM_FIELD_CPU_MASK;
    params.open_mode = UCT_IFACE_OPEN_MODE_DEVICE;
    params.mode.device.tl_name = tl_name;
    params.mode.device.dev_name = dev_name;
    params.stats_root = NULL;
    params.rx_headroom = sizeof(recv_desc_t);
    UCS_CPU_ZERO(&params.cpu_mask);
    /* Read transport-specific interface configuration */
    status = uct_md_iface_config_read(iface_p->md, tl_name, NULL, NULL, &config);
    CHKERR_JUMP(UCS_OK != status, "setup iface_config", error_ret);
    /* Open communication interface */
    assert(iface_p->iface == NULL);
    status = uct_iface_open(iface_p->md, iface_p->worker, &params, config,
(iface_p->iface);
    uct_config_release(config);
    CHKERR_JUMP(UCS_OK != status, "open temporary interface", error_iface);
    /* Enable progress on the interface */
    uct_iface_progress_enable(iface_p->iface, UCT_PROGRESS_SEND | UCT_PROGRESS_RECV);
    /* Get interface attributes */
    status = uct_iface_query(iface_p->iface, &iface_p->iface_attr);
    CHKERR_JUMP(UCS_OK != status, "query iface", error_iface);
    /* Check if current device and transport support required active messages */
    if ((func_am_type == FUNC_AM_SHORT) &&
(iface_p->iface_attr.cap.flags & UCT_IFACE_FLAG_AM_SHORT)) {
        if (test_mem_type != UCS_MEMORY_TYPE_CUDA) {
            return UCS_OK;
        } else {
            fprintf(stderr, "AM short protocol doesn't support CUDA memory");
        }
    }
if ((func_am_type == FUNC_AM_BCOPY) &&
    (iface_p->iface_attr.cap.flags & UCT_IFACE_FLAG_AM_BCOPY)) {
    return UCS_OK;
}
if ((func_am_type == FUNC_AM_ZCOPY) &&
    (iface_p->iface_attr.cap.flags & UCT_IFACE_FLAG_AM_ZCOPY)) {
    return UCS_OK;
}
error_iface:
    uct_iface_close(iface_p->iface);
    iface_p->iface = NULL;
error_ret:
    return UCS_ERR_UNSUPPORTED;
}
/* Device and transport to be used are determined by minimum latency */
static ucs_status_t dev_tl_lookup(const cmd_args_t *cmd_args,
    iface_info_t *iface_p)
{
    uct_tl_resource_desc_t *tl_resources = NULL; /* Communication resource descriptor */
    unsigned num_tl_resources = 0; /* Number of transport resources resource objects */
    uct_component_h *components;
    unsigned num_components;
    unsigned cmpt_index;
    uct_component_attr_t component_attr;
    unsigned md_index;
    unsigned tl_index;
    uct_md_config_t *md_config;
    ucs_status_t status;
    status = uct_query_components(&components, &num_components);
    CHKERR_JUMP(UCS_OK != status, "query for components", error_ret);
    for (cmpt_index = 0; cmpt_index < num_components; ++cmpt_index) {
        component_attr.field_mask = UCT_COMPONENT_ATTR_FIELD_MD_RESOURCE_COUNT;
        status = uct_component_query(components[cmpt_index], &component_attr);
        CHKERR_JUMP(UCS_OK != status, "query component attributes", release_component_list);
        component_attr.field_mask = UCT_COMPONENT_ATTR_FIELD_MD_RESOURCES;
        component_attr.md_resources = alloca(sizeof(*component_attr.md_resources) *
            component_attr.md_resource_count);
        status = uct_component_query(components[cmpt_index], &component_attr);
        CHKERR_JUMP(UCS_OK != status, "query for memory domain resources", release_component_list);
        iface_p->iface = NULL;
        /* Iterate through memory domain resources */
        for (md_index = 0; md_index < component_attr.md_resource_count; ++md_index) {
            status = uct_md_config_read(components[cmpt_index], NULL, NULL,
                &md_config);
            CHKERR_JUMP(UCS_OK != status, "read MD config", release_component_list);
            status = uct_md_open(components[cmpt_index],
                component_attr.md_resources[md_index].md_name,
                md_config, iface_p->md);
            uct_config_release(md_config);
            CHKERR_JUMP(UCS_OK != status, "open memory domains", release_component_list);
            status = uct_md_query(iface_p->md, iface_p->md_attr);
            CHKERR_JUMP(UCS_OK != status, "query iface", close_md);
            status = uct_md_query_tl_resources(iface_p->md, &tl_resources,
                &num_tl_resources);
            CHKERR_JUMP(UCS_OK != status, "query transport resources", close_md);
            /* Go through each available transport and find the proper name */
            for (tl_index = 0; tl_index < num_tl_resources; ++tl_index) {
                if (!strcmp(cmd_args->dev_name, tl_resources[tl_index].dev_name) &&
                    !strcmp(cmd_args->tl_name, tl_resources[tl_index].tl_name)) {
                    fprintf(stderr, "Unsupported memory type \n",
                        UCT_TL_RESOURCE_DESC_FMT " on %s MD\n",
                        ucs_memory_type_names[test_mem_type],
                        UCT_TL_RESOURCE_DESC_ARG(tl_resources[tl_index]),
                        component_attr.md_resources[md_index].md_name);
                    status = UCS_ERR_UNSUPPORTED;
                    break;
                }
            }
            status = init_iface(tl_resources[tl_index].dev_name,
                tl_resources[tl_index].tl_name,
                cmd_args->func_am_type, iface_p);
            if (status != UCS_OK) {
                break;
            }
            fprintf(stdout, "Using "UCT_TL_RESOURCE_DESC_FMT" on\n",
                UCT_TL_RESOURCE_DESC_ARG(tl_resources[tl_index]));
            goto release_tl_resources;
        }
    }
    release_tl_resources:
        uct_release_tl_resource_list(tl_resources);
if ((status == UCS_OK) &&
    (tl_index < num_tl_resources)) {
    goto release_component_list;
}
    tl_resources = NULL;
    num_tl_resources = 0;
    uct_md_close(iface_p->md);
}
)
fprintf(stderr, "No supported (dev/tl) found (%s/%s)\n",
    cmd_args->dev_name, cmd_args->tl_name);
    status = UCS_ERR_UNSUPPORTED;
release_component_list:
    uct_release_component_list(components);
error_ret:
    return status;
close_md:
    uct_md_close(iface_p->md);
    goto release_component_list;
}
int print_err_usage()
{
    const char func_template[] = " -%c Select \"%s\" function to send the message%sn"
    fprintf(stderr, "%" 
    fprintf(stderr, "Usage: uct_hello_world [parameters]\n"
    fprintf(stderr, "
    fprintf(stderr, func_template, 'i', func_am_t_str(FUNC_AM_SHORT), " (default)"');
    fprintf(stderr, func_template, 'b', func_am_t_str(FUNC_AM_BCOPY), "'"');
    fprintf(stderr, func_template, 'z', func_am_t_str(FUNC_AM_ZCOPY), ");
    fprintf(stderr, " -d Select device name\n"
    fprintf(stderr, func_template, 't', func_am_t_str(FUNC_AM_TCP), "'"');
    print_commonHelp();
    fprintf(stderr, "%" 
    fprintf(stderr, "Server: uct_hello_world -d eth0 -t tcp\n"
    fprintf(stderr, "Client: uct_hello_world -d eth0 -t tcp -n localhost\n"
    return UCS_ERR_UNSUPPORTED;
}
int parse_cmd(int argc, char * const argv[], cmd_args_t *args)
{
    int c = 0, idx = 0;
    assert(args);
    memset(args, 0, sizeof(*args));
    /* Defaults */
    args->server_port = 13337;
    args->func_am_type = FUNC_AM_SHORT;
    args->test_strlen = 16;
    opterr = 0;
    while ((c = getopt(argc, argv, "ibzd:t:n:p:s:m:h")) != -1) {
switch (c) {
    case 'i':
        args->func_am_type = FUNC_AM_SHORT;
        break;
    case 'b':
        args->func_am_type = FUNC_AM_BCOPY;
        break;
    case 'z':
        args->func_am_type = FUNC_AM_ZCOPY;
        break;
    case 'd':
        args->dev_name = optarg;
        break;
    case 't':
        args->tl_name = optarg;
        break;
    case 'n':
        args->server_name = optarg;
        break;
    case 'p':
        args->server_port = atoi(optarg);
        if (args->server_port <= 0)
            fprintf(stderr, "Wrong server port number %d\n",
                args->server_port);
        return UCS_ERR_UNSUPPORTED;
    break;
    case 's':
        args->test_strlen = atol(optarg);
        if (args->test_strlen <= 0)
            fprintf(stderr, "Wrong string size %ld\n", args->test_strlen);
        return UCS_ERR_UNSUPPORTED;
    break;
    case 'm':
        test_mem_type = parse_mem_type(optarg);
        if (test_mem_type == UCS_MEMORY_TYPE_LAST)
            return UCS_ERR_UNSUPPORTED;
    break;
    default:
        fprintf(stderr, "Unknown option \"%c\"\n"
            return UCS_ERR_UNSUPPORTED;
    break;
}}}
break;

case 'g':
  if (optopt == '-') { 
    fprintf(stderr, "Option -- requires an argument.\n", optopt);
  } else if (isprint (optopt)) {
    fprintf(stderr, "Option argument \"-%s\".\n", optopt);
  } else {
    fprintf(stderr, "Unknown option character \"\%c\\".\n", optopt);
  }

case 'h':
  default:
    return print_err_usage();
  }

fprintf(stderr, "INFO: UCT_HELLO_WORLD AM function = %s server = %s port = %d\n",
        func_am_t_str(args->func_am_type), args->server_name,
        args->server_port);
for (idx = optind; idx < argc; idx++) {
  fprintf(stderr, "WARNING: Non-option argument %s\n", argv[idx]);
}
if (args->dev_name == NULL) {
  fprintf(stderr, "WARNING: Non-option argument %s", argv[idx]);
  return print_err_usage();
}
if (args->tl_name == NULL) {
  fprintf(stderr, "WARNING: transport layer is not set\n!");
  return print_err_usage();
}
return UCS_OK;
}

int sendrecv(int sock, const void *sbuf, size_t slen, void **rbuf) {
  int ret = 0;
  size_t rlen = 0;
  *rbuf = NULL;
  ret = send(sock, sbuf, slen, 0);
  if (ret != (int)slen) {
    fprintf(stderr, "failed to send buffer, return value %d\n", ret);
    return -1;
  }
  size_t slen = ret;
  ret = recv(sock, rbuf, rlen, MSG_WAITALL);
  if (ret != (int)rlen) {
    fprintf(stderr, "failed to receive device address, return value %d\n", ret);
    return -1;
  }
  *rbuf = calloc(1, rlen);
  if (!*rbuf) {
    fprintf(stderr, "failed to allocate receive buffer\n", ret);
    return -1;
  }
  ret = recv(sock, sbuf, slen, 0);
  if (ret != (int)slen) {
    fprintf(stderr, "failed to receive buffer length\n", ret);
    return -1;
  }
}

int main(int argc, char **argv) {
  uct_device_addr_t peer_dev = NULL;
  uct_iface_addr_t peer_iface = NULL;
  uct_ep_addr_t own_ep = NULL;
  uct_ep_addr_t peer_ep = NULL;
  uint8_t id = 0;
  int sock = -1; /* OOB connection socket */
  ucs_status_t status = UCS_OK; /* status codes for UCS */
  uct_device_addr_t own_dev;
  uct_iface_addr_t own_iface;
  uct_ep_h ep; /* Remote endpoint */
  ucs_async_context_t *async; /* Async event context manages times and fd notifications */
  cmd_args_t cmd_args;
  iface_info_t if_info;
  uct_ep_params_t ep_params;
  int res;
  /* Parse the command line */
  if (parse_cmd(argc, argv, &cmd_args)) {
    status = UCS_ERR_INVALID_PARAM;
  }
  /* The caller is responsible to free *rbuf */
  return 0;
}
goto out;
}

/* Initialize context */
/* It is better to use different contexts for different workers */
status = ucs_async_context_create(UCS_ASYNC_MODE_THREAD_SPINLOCK, async);
CHECKERR_JUMP((status != UCS_OK), "init async context", out);
/* Create a worker object */
status = uct_worker_create(async, UCS_THREAD_MODE_SINGLE, &if_info.worker);
CHECKERR_JUMP((status != UCS_OK), "create worker", out_cleanup_async);
/* Search for the desired transport */
status = dev_tl_lookup(&cmd_args, &if_info);
CHECKERR_JUMP((status != UCS_OK), "find supported device and transport", out_free_dev_addrs);

/* Get device address */
status = uct_iface_get_device_address(if_info.iface, own_dev);
CHECKERR_JUMP((status != UCS_OK), "get device address", out_free_if_addrs);
if (cmd_args.server_name) {
    oob_sock = client_connect(cmd_args.server_name, cmd_args.server_port);
} else {
    oob_sock = server_connect(cmd_args.server_port);
}
CHECKERR_ACTION((oob_sock < 0), "OOB connect", status = UCS_ERR_IO_ERROR; goto out_close_oob_sock);
res = sendrecv(oob_sock, own_dev, if_info.iface_attr.device_addr_len, (void **)&peer_dev);
CHECKERR_ACTION((0 != res), "device exchange", status = UCS_ERR_NO_MESSAGE; goto out_close_oob_sock);
status = (ucs_status_t)uct_iface_is_reachable(if_info.iface, peer_dev, NULL);
CHECKERR_JUMP((0 == status), "reach the peer", out_close_oob_sock);
/* Get interface address */
if (if_info.iface_attr.cap.flags & UCT_IFACE_FLAG_CONNECT_TO_IFACE) {
    status = uct_iface_get_address(if_info.iface, own_iface);
    CHKERR_JUMP((status != UCS_OK), "get interface address", out_close_oob_sock);
    status = (ucs_status_t)sendrecv(oob_sock, own_iface, if_info.iface_attr.iface_addr_len, (void **)&peer_iface);
    CHKERR_JUMP((0 != status), "ifaces exchange", out_close_oob_sock);
}
ep_params.field_mask = UCT_EP_PARAM_FIELD_IFACE;
ep_params.iface = if_info.iface;
if (if_info.iface_attr.cap.flags & UCT_IFACE_FLAG_CONNECT_TO_EP) {
    own_ep = (uct_ep_addr_t*)calloc(1, if_info.iface_attr.ep_addr_len);
    CHKERR_ACTION((NULL == own_ep), "allocate memory for ep addrs", status = UCS_ERR_NO_MEMORY; goto out_close_oob_sock);
    /* Create new endpoint */
    status = uct_ep_create(&ep_params, &ep);
    CHKERR_JUMP((status != UCS_OK), "create endpoint", out_free_ep_addrs);
    /* Get endpoint address */
    status = uct_ep_get_address(ep, own_ep);
    CHKERR_JUMP((status != UCS_OK), "get endpoint address", out_free_ep);
    status = (ucs_status_t)sendrecv(oob_sock, own_ep, if_info.iface_attr.ep_addr_len, (void **)&peer_ep);
    CHKERR_JUMP((0 != status), "EPs exchange", out_free_ep);
    /* Connect endpoint to a remote endpoint */
    status = uct_ep_connect_to_ep(ep, peer_dev, peer_ep);
    if (barrier(oob_sock)) {
        status = UCS_ERR_IO_ERROR;
        goto out_free_ep;
    }
}
else if (if_info.iface_attr.cap.flags & UCT_IFACE_FLAG_CONNECT_TO_IFACE) {
    /* Create an endpoint which is connected to a remote interface */
    ep_params.dev_addr = peer_dev;
    ep_params.iface_addr = peer_iface;
    status = uct_ep_create(&ep_params, &ep);
    CHKERR_JUMP((status != UCS_OK), "create endpoint", out_free_ep_addrs);
    /* Set active message handler */
    status = uct_iface_set_am_handler(if_info.iface, id, hello_world, &cmd_args.func_am_type, 0);
}
else {
    status = UCS_ERR_UNSUPPORTED;
    goto out_free_ep_addrs;
}
if (cmd_args.test_strlen > func_am_max_size(&cmd_args.func_am_type, &if_info.iface_attr)) {
    status = UCS_ERR_UNSUPPORTED;
    fprintf(stderr, "Test string is too long: %ld, max supported: %lu\n", cmd_args.test_strlen,
            func_am_max_size(&cmd_args.func_am_type, &if_info.iface_attr));
    goto out_free_ep;
}
/* Set active message handler */
status = uct_iface_set_am_handler(if_info.iface, id, hello_world, &cmd_args.func_am_type, 0);
CHERR_JUMP(UCS_OK != status, "set callback", out_free_ep);
if (cmd_args.server_name) {
    char *str = (char *)mem_type_malloc(cmd_args.test_strlen);
    CHKERR_ACTION(str == NULL, "allocate memory",
        status = UCS_ERR_NO_MEMORY; goto out_free_ep);
    res = generate_test_string(str, cmd_args.test_strlen);
    CHKERR_ACTION(res < 0, "generate test string",
        status = UCS_ERR_NO_MEMORY; goto out_free_ep);
    /* Send active message to remote endpoint */
    if (cmd_args.func_am_type == FUNC_AM_SHORT) {
        status = do_am_short(&if_info, ep, id, &cmd_args, str);
    } else if (cmd_args.func_am_type == FUNC_AM_BCOPY) {
        status = do_am_bcopy(&if_info, ep, id, &cmd_args, str);
    } else if (cmd_args.func_am_type == FUNC_AM_ZCOPY) {
        status = do_am_zcopy(&if_info, ep, id, &cmd_args, str);
    }
    mem_type_free(str);
    CHKERR_JUMP(UCS_OK != status, "send active msg", out_free_ep);
} else {
    recv_desc_t *rdesc;
    while (desc_holder == NULL) {
        /* Explicitly progress any outstanding active message requests */
        uct_worker_progress(if_info.worker);
    }
    rdesc = desc_holder;
    print_strings("main", func_am_t_str(cmd_args.func_am_type),
        (char *)(rdesc + 1), cmd_args.test_strlen);
    if (rdesc->is_uct_desc) {
        /* Release descriptor because callback returns UCS_INPROGRESS */
        uct_iface_release_desc(rdesc);
    } else {
        free(rdesc);
    }
}
if (barrier(oob_sock)) {
    status = UCS_ERR_IO_ERROR;
}
out_free_ep:
    uct_ep_destroy(ep);
out_free_ep_addrs:
    free(own_ep);
    free(peer_ep);
out_close_oob_sock:
    close(oob_sock);
out_free_if_addrs:
    free(own_iface);
    free(peer_iface);
out_free_dev_addrs:
    free(own_dev);
    free(peer_dev);
out_destroy_iface:
    uct_iface_close(if_info.iface);
    uct_md_close(if_info.md);
out_destroy_worker:
    uct_worker_destroy(if_info.worker);
out_cleanup_async:
    ucs_async_context_destroy(async);
out:
    return (status == UCS_ERR_UNSUPPORTED) ? UCS_OK : status;
Index

completed_cb
  uct_tag_context, 228

finish
  UCP Data type routines, 120

pack
  UCP Data type routines, 120
packed_size
  UCP Data type routines, 119
priv
  uct_tag_context, 229

rndv_cb
  uct_tag_context, 228

start_pack
  UCP Data type routines, 119
start_unpack
  UCP Data type routines, 119
tag_consumed_cb
  uct_tag_context, 228

UCP Application Context, 12
  UCP_ATTR_FIELD_MEMORY_TYPES, 17
  UCP_ATTR_FIELD_REQUEST_SIZE, 17
  ucp_cleanup, 18
  ucp_context_attr_field, 17
  ucp_context_h, 15
  ucp_context_print_info, 19
  ucp_context_query, 19
  ucp_feature, 16
  UCP_FEATURE_AM, 17
  UCP_FEATURE_AMO32, 17
  UCP_FEATURE_AMO64, 17
  UCP_FEATURE_RMA, 17
  UCP_FEATURE_STREAM, 17
  UCP_FEATURE_TAG, 17
  UCP_FEATURE_WAKEUP, 17
  ucp_get_version, 17
  ucp_get_version_string, 17
  ucp_init, 17
  UCP_PARAM_FIELD_ESTIMATED_NUM_EPS, 16
  UCP_PARAM_FIELD_ESTIMATED_NUM_PPN, 16
  UCP_PARAM_FIELD_FEATURES, 16
  UCP_PARAM_FIELD_MT_WORKERS_SHARED, 16
  UCP_PARAM_FIELD_REQUEST_CLEANUP, 16
  UCP_PARAM_FIELD_REQUEST_INIT, 16
  UCP_PARAM_FIELD_REQUEST_SIZE, 16
  ucp_params_field, 16
  ucp_request_cleanup_callback_t, 16
  ucp_request_init_callback_t, 15
  ucp_tag_recv_info_t, 15
  UCP Communication routines, 68
    ucp_am_data_release, 80
    ucp_am_handler_param_field, 77
    UCP_AM_HANDLER_PARAM_FIELD_ARG, 78
    UCP_AM_HANDLER_PARAM_FIELD_CB, 78
    UCP_AM_HANDLER_PARAM_FIELD_FLAGS, 78
    UCP_AM_HANDLER_PARAM_FIELD_ID, 77
    UCP_AM_RECV_ATTR_FIELD_REPLY_EP, 77
    UCP_AM_RECV_ATTR_FLAG_DATA, 77
    UCP_AM_RECV_ATTR_FLAG_RNDV, 77
    ucp_am_recv_attr_t, 77
    ucp_am_recv_data_nbx, 79
    ucp_am_recv_data_nbx_callback_t, 75
    ucp_am_send_nb, 78
    ucp_am_send_nbx, 78
    ucp_atomic_add32, 103
    ucp_atomic_add64, 104
    ucp_atomic_cswap32, 108
    ucp_atomic_cswap64, 109
    ucp_atomic_fetch_nb, 97
    UCP_ATOMIC_FETCH_OP_CSWAP, 76
    UCP_ATOMIC_FETCH_OP_FADD, 76
    UCP_ATOMIC_FETCH_OP_FAND, 76
    UCP_ATOMIC_FETCH_OP_FOR, 76
    UCP_ATOMIC_FETCH_OP_FXOR, 76
    UCP_ATOMIC_FETCH_OP_LAST, 76
    UCP_ATOMIC_FETCH_OP_SWAP, 76
    ucp_atomic_fetch_op_nbx, 98
    UCP_ATOMIC_FETCH_OP_SWAP, 76
    ucp_atomic_fetch_op_t, 75
    UCP_ATOMIC_OP_ADD, 76
    UCP_ATOMIC_OP_AND, 76
    UCP_ATOMIC_OP_CSWAP, 76
    UCP_ATOMIC_OP_LAST, 76
    UCP_ATOMIC_OP_OR, 76
    UCP_ATOMIC_OP_SWAP, 76
    ucp_atomic_op_nbx, 98
    UCP_ATOMIC_OP_SWAP, 76
    ucp_atomic_op_t, 76
    UCP_ATOMIC_OP_XOR, 76
ucp_atomic_post, 97
UCP_ATOMIC_POST_OP_ADD, 75
UCP_ATOMIC_POST_OP_AND, 75
UCP_ATOMIC_POST_OP_LAST, 75
UCP_ATOMIC_POST_OP_OR, 75
ucp_atomic_post_op_t, 75
UCP_ATOMIC_POST_OP_XOR, 75
ucp_atomic_swap32, 106
ucp_atomic_swap64, 107
ucp_datatype_t, 72
ucp_err_handler_cb_t, 73
ucp_err_handler_t, 73
ucp_get, 103
ucp_get_nb, 95
ucp_get_nbi, 95
ucp_get_nbx, 96
UCP_OP_ATTR_FIELD_CALLBACK, 77
UCP_OP_ATTR_FIELD_DATATYPE, 77
UCP_OP_ATTR_FIELD_FLAGS, 77
UCP_OP_ATTR_FIELD_MEMORY_TYPE, 77
UCP_OP_ATTR_FIELD_RECV_INFO, 77
UCP_OP_ATTR_FIELD_REPLY_BUFFER, 77
UCP_OP_ATTR_FIELD_REQUEST, 76
UCP_OP_ATTR_FIELD_USER_DATA, 77
UCP_OP_ATTR_FLAG_FAST_CMPL, 77
UCP_OP_ATTR_FLAG_FORCE_IMM_CMPL, 77
UCP_OP_ATTR_FLAG_NO_IMM_CMPL, 77
ucp_op_attr_t, 76
ucp_put, 102
ucp_put_nb, 93
ucp_put_nbi, 92
ucp_put_nbx, 94
ucp_request_alloc, 101
ucp_request_cancel, 100
ucp_request_check_status, 99
ucp_request_free, 101
ucp_request_is_completed, 102
ucp_send_callback_t, 72
ucp_send_nbx_callback_t, 72
ucp_stream_data_release, 101
ucp_stream_recv_callback_t, 73
ucp_stream_recv_data_nb, 87
UCP_STREAM_RECV_FLAG_WAITALL, 76
ucp_stream_recv_flags_t, 76
ucp_stream_recv_nb, 86
ucp_stream_recv_nbx, 87
ucp_stream_recv_nbx_callback_t, 74
ucp_stream_recv_request_test, 100
ucp_stream_send_nb, 80
ucp_stream_send_nbx, 81
ucp_tag_message_h, 72
ucp_tag_msg_recv_nb, 91
ucp_tag_msg_recv_nbx, 92
ucp_tag_msg_nbx, 90
ucp_tag_msg_recv_callback_t, 74
ucp_tag_msg_recv_nb, 88
ucp_tag_msg_recv_nbr, 89
ucp_tag_msg_recv_nbx, 89
ucp_tag_recv_nb_callback_t, 74
ucp_tag_recv_request_test, 100
ucp_tag_send_nb, 82
ucp_tag_send_nbx, 84
ucp_tag_send_nbx_callback_t, 74
ucp_tag_send_nbx_callback, 85
ucp_tag_t, 72
UCP Configuration, 110
ucp_config_modify, 113
ucp_config_print, 114
ucp_config_read, 112
ucp_config_release, 113
ucp_config_t, 112
ucp_params_t, 112
UCP Data type routines, 115
finish, 120
pack, 120
packed_size, 119
start_pack, 119
start_unpack, 119
UCP_DATATYPE_CLASS_MASK, 118
UCP_DATATYPE_CONTIG, 118
UCP_DATATYPE_GENERIC, 118
UCP_DATATYPE_IOV, 118
UCP_DATATYPE_SHIFT, 118
UCP_DATATYPE_STRIDED, 118
ucp_dt_create_generic, 118
ucp_dt_destroy, 118
ucp_dt_ipv_t, 117
ucp_dt_make_contig, 116
ucp_dt_make_ipv, 117
ucp_dt_type, 117
ucp_dt_type, 117
ucp_generic_dt_ops_t, 117
unpack, 120
UCP Endpoint, 56
ucp_am_callback_t, 59
ucp_am_recv_callback_t, 59
UCP_CB_PARAM_FLAG_DATA, 62
ucp_cb_param_flags, 61
ucp_conn_request_h, 58
ucp_disconnect_nb, 66
UCP_EP_CLOSE_FLAG_FORCE, 61
ucp_ep_close_flags_t, 61
ucp_ep_close_mode, 61
UCP_EP_CLOSE_MODE_FLUSH, 61
UCP_EP_CLOSE_MODE_FORCE, 61
ucp_ep_close_nb, 63
ucp_ep_close_nbx, 63
ucp_ep_create, 62
ucp_ep_destroy, 66
ucp_ep_flush, 66
ucp_ep_flush_nb, 64
ucp_ep_flush_nbx, 65
ucp_ep_t, 58
ucp_ep_type, 58
ucp_ep_type, 58
ucp_param_type, 58
UCP_EP_PARAM_FIELD_CONN_REQUEST, 60
UCP_EP_PARAM_FIELD_ERR_HANDLER, 60
ucp_ep_close_mode, 61
ucp_ep_close_mode, 61
ucp_ep_close_mode, 61
ucp_ep_close_mode, 61
ucp_ep_create, 62
ucp_ep_destroy, 66
ucp_ep_flush, 66
ucp_ep_flush_nb, 64
ucp_ep_flush_nbx, 65
ucp_ep_t, 58
UCP_ATOMIC_OP_ADD
UCP Communication routines, 76
UCP_ATOMIC_OP_AND
UCP Communication routines, 76
UCP_ATOMIC_OP_CSWAP
UCP Communication routines, 76
UCP_ATOMIC_OP_LAST
UCP Communication routines, 76
ucp_atomic_op_nbx
UCP Communication routines, 98
UCP_ATOMIC_OP_OR
UCP Communication routines, 76
UCP_ATOMIC_OP_SWAP
UCP Communication routines, 76
ucp_atomic_op_t
UCP Communication routines, 76
UCP_ATOMIC_OP_XOR
UCP Communication routines, 76
ucp_atomic_post
UCP Communication routines, 97
UCP_ATOMIC_POST_OP_ADD
UCP Communication routines, 75
UCP_ATOMIC_POST_OP_AND
UCP Communication routines, 75
UCP_ATOMIC_POST_OP_LAST
UCP Communication routines, 75
UCP_ATOMIC_POST_OP_OR
UCP Communication routines, 75
ucp_atomic_post_op_t
UCP Communication routines, 75
UCP_ATOMIC_POST_OP_XOR
UCP Communication routines, 75
ucp_atomic_swap32
UCP Communication routines, 106
ucp_atomic_swap64
UCP Communication routines, 107
UCP_ATTR_FIELD_MEMORY_TYPES
UCP Application Context, 17
UCP_ATTR_FIELD_REQUEST_SIZE
UCP Application Context, 17
UCP_ATTR_FIELD_THREAD_MODE
UCP Application Context, 17
UCP_CONN_REQUEST_ATTR_FIELD_CLIENT_ADDR
UCP Worker, 30
ucp_conn_request_attr
UCP Worker, 30
ucp_conn_request_attr_field
UCP Worker, 30
UCP_CONN_REQUEST_ATTR_FIELD_CLIENT_ADDR
UCP Worker, 30
ucp_conn_request_attr_t
UCP Worker, 30
ucp_conn_request_h
UCP Endpoint, 58
ucp_conn_request_query
UCP Worker, 36
ucp_context_attr
UCP Application Context, 15
ucp_context_attr_field
UCP Application Context, 15
ucp_context_h
UCP Application Context, 15
ucp_context_print_info
UCP Application Context, 19
ucp_context_query
UCP Application Context, 19
UCP_DATATYPE_CLASS_MASK
UCP Data type routines, 118
UCP_DATATYPE_CONTIG
UCP Data type routines, 118
UCP_DATATYPE_GENERIC
UCP Data type routines, 118
UCP_DATATYPE_IOV
UCP Data type routines, 118
UCP_DATATYPE_SHIFT
UCP Data type routines, 118
UCP_DATATYPE_STRIDED
UCP Data type routines, 118
ucp_datatype_t
UCP Communication routines, 72
ucp_disconnect_nb
UCP Endpoint, 66
ucp_dt_create_generic
UCP Data type routines, 118
ucp_dt_destroy
UCP Data type routines, 118
ucp_dt_make_contig
UCP Data type routines, 118
ucp_dt_make_iov
UCP Data type routines, 118
ucp_dt_type
UCP Data type routines, 118
ucp_dt iov, 116
ucp_dt iov_t
UCP Data type routines, 117
ucp_dt_make_contig
UCP Data type routines, 116
ucp_dt_make_iov
UCP Data type routines, 117
ucp_dt_type
UCP Data type routines, 117
UCP_EP_CLOSE_FLAG_FORCE
UCP Endpoint, 61
ucp_ep_close_flags_t
UCP Endpoint, 61
ucp_ep_close_mode
UCP Endpoint, 61
UCP_EP_CLOSE_MODE_FLUSH
UCP Endpoint, 61
UCP_EP_CLOSE_MODE_FORCE
UCP Endpoint, 61
ucp_ep_close_nb
UCP Endpoint, 63
ucp_ep_close_nbx
UCP Endpoint, 63
ucp_ep_create
UCP Endpoint, 62
ucp_ep_destroy
UCP Endpoint, 66
ucp_ep_flush
UCP Endpoint, 66
ucp_ep_flush_nb
UCP Endpoint, 64
ucp_ep_flush_nbx
UCP Endpoint, 65
ucp_ep_h
UCP Endpoint, 58
ucp_ep_modify_nb
UCP Endpoint, 66
UCP_EP_PARAM_FIELD_CONN_REQUEST
UCP Endpoint, 60
UCP_EP_PARAM_FIELD_ERR_HANDLER
UCP Endpoint, 60
UCP_EP_PARAM_FIELD_ERR_HANDLING_MODE
UCP Endpoint, 60
UCP_EP_PARAM_FIELD_FLAGS
UCP Endpoint, 60
UCP_EP_PARAM_FIELD_REMOTE_ADDRESS
UCP Endpoint, 60
UCP_EP_PARAM_FIELD_SOCK_ADDR
UCP Endpoint, 60
UCP_EP_PARAM_FIELD_USER_DATA
UCP Endpoint, 60
ucp_ep_params, 57
ucp_ep_params_field
UCP Endpoint, 60
UCP_EP_PARAMS_FLAGS_CLIENT_SERVER
UCP Endpoint, 61
ucp_ep_params_flags_field
UCP Endpoint, 60
UCP_EP_PARAMS_FLAGS_NO_LOOPBACK
UCP Endpoint, 61
ucp_ep_params_t
UCP Endpoint, 61
ucp_ep_print_info
UCP Endpoint, 64
ucp_ep_rkey_unpack
UCP Memory routines, 50
ucp_err_handler, 72
ucp_err_handler_cb_t
UCP Communication routines, 73
ucp_err_handler_t
UCP Communication routines, 73
UCP_ERR_HANDLING_MODE_NONE
UCP Endpoint, 62
UCP_ERR_HANDLING_MODE_PEER
UCP Endpoint, 62
ucp_err_handling_mode_t
UCP Endpoint, 62
ucp_feature
UCP Application Context, 16
UCP_FEATURE_AM
UCP Application Context, 17
UCP_FEATURE_AMO32
UCP Application Context, 17
UCP_FEATURE_AMO64
UCP Application Context, 17
UCP_FEATURE_RMA
UCP Application Context, 17
UCP_FEATURE_STREAM
UCP Application Context, 17
UCP_FEATURE_TAG
UCP Application Context, 17
UCP_FEATURE_WAKEUP
UCP Application Context, 17
ucp_generic_dt_ops, 227
ucp_generic_dt_ops_t
UCP Data type routines, 117
ucp_get
UCP Communication routines, 103
ucp_get_nb
UCP Communication routines, 95
ucp_get_nbi
UCP Communication routines, 95
ucp_get_nbx
UCP Communication routines, 96
ucp_get_version
UCP Application Context, 17
ucp_get_version_string
UCP Application Context, 17
ucp_init
UCP Application Context, 17
ucp_listener_accept_callback_t
UCP Worker, 28
ucp_listener_accept_handler, 25
ucp_listener_accept_handler_t
UCP Worker, 27
ucp_listener_attr, 24
ucp_listener_attr_field
UCP Worker, 30
UCP_LISTENER_ATTR_FIELD_SOCKADDR
UCP Worker, 30
ucp_listener_attr_t
UCP Worker, 26
ucp_listener_conn_callback_t
UCP Worker, 28
ucp_listener_conn_handler, 26
ucp_listener_conn_handler_t
UCP Worker, 28
ucp_listener_create
UCP Worker, 35
ucp_listener_destroy
UCP Worker, 36
ucp_listener_h
UCP Communication routines, 84
ucp_tag_send_sync_nbx
UCP Communication routines, 85
ucp_tag_t
UCP Communication routines, 72
UCP_WAKEUP_AMO
UCP Worker, 31
UCP_WAKEUP_EDGE
UCP Worker, 31
ucp_wakeup_event_t
UCP Worker, 28
ucp_wakeup_event_types
UCP Worker, 31
UCP_WAKEUP_RMA
UCP Worker, 31
UCP_WAKEUP_RX
UCP Worker, 31
UCP_WAKEUP_TAG_RECV
UCP Worker, 31
UCP_WAKEUP_TAG_SEND
UCP Worker, 31
UCP_WAKEUP_TX
UCP Worker, 31
UCP_WORKER_ADDRESS_FLAG_NET_ONLY
UCP Worker, 30
ucp_worker_address_flags_t
UCP Worker, 29
ucp_worker_arm
UCP Wake-up routines, 54
ucp_worker_attr
UCP Worker, 22
ucp_worker_attr_field
UCP Worker, 30
UCP_WORKER_ATTR_FIELD_ADDRESS
UCP Worker, 30
UCP_WORKER_ATTR_FIELD_ADDRESS_FLAGS
UCP Worker, 30
UCP_WORKER_ATTR_FIELD_MAX_AM_HEADER
UCP Worker, 30
UCP_WORKER_ATTR_FIELD_THREAD_MODE
UCP Worker, 30
ucp_worker_attr_t
UCP Worker, 26
ucp_worker_create
UCP Worker, 32
ucp_worker_destroy
UCP Worker, 32
ucp_worker_fence
UCP Worker, 38
ucp_worker_flush
UCP Worker, 40
ucp_worker_flush_nb
UCP Worker, 39
ucp_worker_flush_nb
UCP Worker, 39
ucp_worker_get_address
UCP Worker, 33
ucp_worker_get_efd
UCP Wake-up routines, 52
cucp_worker_h
UCP Worker, 27
UCP_WORKER_PARAM_FIELD_CPU_MASK
UCP Worker, 29
UCP_WORKER_PARAM_FIELD_EVENT_FD
UCP Worker, 29
UCP_WORKER_PARAM_FIELD_EVENTS
UCP Worker, 29
UCP_WORKER_PARAM_FIELD_THREAD_MODE
UCP Worker, 29
UCP_WORKER_PARAM_FIELD_USER_DATA
UCP Worker, 29
ucp_worker_params
UCP Worker, 23
ucp_worker_params_field
UCP Worker, 29
ucp_worker_params_t
UCP Worker, 26
ucp_worker_print_info
UCP Worker, 37
ucp_worker_progress
UCP Worker, 34
ucp_worker_query
UCP Worker, 33
ucp_worker_release_address
UCP Worker, 34
ucp_worker_set_am_handler
UCP Worker, 37
ucp_worker_set_am_recv_handler
UCP Worker, 38
ucp_worker_signal
UCP Wake-up routines, 55
ucp_worker_wait
UCP Wake-up routines, 53
ucp_worker_wait_mem
UCP Wake-up routines, 53
UCS Communication Resource, 219
ucs_async_add_timer
UCS Worker, 223
ucs_async_context_create
UCS Worker, 225
ucs_async_context_destroy
UCS Worker, 225
ucs_async_event_cb_t
UCS Worker, 220
ucs_async_modify_handler
UCS Worker, 224
ucs_async_poll
UCS Worker, 225
ucs_async_remove_handler
UCS Worker, 224
ucs_async_set_event_handler
UCS Worker, 223
UCS_CALLBACKQ_FLAG_FAST
UCS Worker, 221
UCS_CALLBACKQ_FLAG_ONESHOT
UCS Worker, 221
ucs_callbackq_flags
UCS Worker, 221
UCS_ERR_ALREADY_EXISTS
UCS Worker, 222
UCS_ERR_BUFFER_TOO_SMALL
UCS Worker, 222
UCS_ERR_BUSY
UCS Worker, 222
UCS_ERR_CANCELED
UCS Worker, 222
UCS_ERR_CONNECTION_RESET
UCS Worker, 222
UCS_ERR_ENDPOINT_TIMEOUT
UCS Worker, 222
UCS_ERR_EXCEEDS_LIMIT
UCS Worker, 222
UCS_ERR_FIRST_ENDPOINT_FAILURE
UCS Worker, 222
UCS_ERR_FIRST_LINK_FAILURE
UCS Worker, 222
UCS_ERR_INVALID_ADDR
UCS Worker, 222
UCS_ERR_INVALID_PARAM
UCS Worker, 222

© 2021 Unified Communication X (UCX). All rights reserved.
UCT_CM_REMOTE_DATA_FIELD_DEV_ADDR_LENGTH, 205
uct_cm_remote_data_t, 201
uct_ep_disconnect, 207
uct_ep_disconnect_cb_t, 203
uct_iface_accept, 206
uct_iface_reject, 207
uct_listener_attr_field, 204
UCT_LISTENER_ATTR_FIELD_SOCKADDR, 204
uct_listener_create, 209
uct_listener_destroy, 210
UCT_LISTENER_PARAM_FIELD_BACKLOG, 204
UCT_LISTENER_PARAM_FIELD_CONN_REQUEST_CB, 204
UCT_LISTENER_PARAM_FIELD_USER_DATA, 204
uct_listener_params_field, 204
uct_listener_query, 210
uct_listener_reject, 210
uct_sockaddr_conn_request_callback_t, 201
UCT Communication Context, 160
UCT_ALLOC_METHOD_DEFAULT, 161
UCT_ALLOC_METHOD_HEAP, 160
UCT_ALLOC_METHOD_HUGE, 161
UCT_ALLOC_METHOD_LAST, 161
UCT_ALLOC_METHOD_MD, 160
UCT_ALLOC_METHOD_MMAP, 160
uct_alloc_method_t, 160
UCT_ALLOC_METHOD_THP, 160
uct_config_get, 162
uct_config_modify, 163
uct_worker_create, 161
uct_worker_destroy, 161
uct_worker_progress, 163
uct_worker_progress_register_safe, 161
uct_worker_progress_unregister_safe, 162
UCT Communication Resource, 122
uct_async_event_cb_t, 137
UCT_CB_FLAG_ASYNC, 143
UCT_CB_FLAG_RESERVED, 143
uct_completion_callback_t, 145
uct_completion_t, 147
uct_completion_update_status, 159
uct_component_attr_field, 141
UCT_COMPONENT_ATTR_FIELD_MD_RESOURCES, 141
UCT_COMPONENT_ATTR_FIELD_NAME, 141
uct_component_attr_t, 135
UCT_COMPONENT_FLAG_CM, 142
uct_component_h, 135
uct_component_query, 146
uct_component_release, 149
uct_conn_request_h, 139
uct_device_addr_t, 137
UCT_DEVICE_TYPE_ACC, 142
UCT_DEVICE_TYPE_LAST, 142
UCT_DEVICE_TYPE_NET, 142
UCT_DEVICE_TYPE_SELF, 142
uct_device_type_t, 142
uct_ep_addr_t, 137
uct_ep_check, 152
uct_ep_connect_to_ep, 155
uct_ep_create, 154
uct_ep_destroy, 154
uct_ep_fence, 157
uct_ep_flush, 157
uct_ep_get_address, 154
uct_ep_h, 136
UCT_EP_PARAM_FIELD_CM, 145
UCT_EP_PARAM_FIELD_CONN_REQUEST, 145
UCT_EP_PARAM_FIELD_DEV_ADDR, 144
UCT_EP_PARAM_FIELD_IFACE, 144
UCT_EP_PARAM_FIELD_IFACE_ADDR, 144
UCT_EP_PARAM_FIELD_PATH_INDEX, 145
UCT_EP_PARAM_FIELD_SOCKADDR, 144
UCT_EP_PARAM_FIELD_SOCKADDR_CB_FLAGS, 145
UCT_EP_PARAM_FIELD_SOCKADDR_CONNECT_CB_CLIENT, 145
UCT_EP_PARAM_FIELD_SOCKADDR_DISCONNECT_CB, 145
UCT_EP_PARAM_FIELD_SOCKADDR_NOTIFY_CB_SERVER, 145
UCT_EP_PARAM_FIELD_SOCKADDR_PACKAGE_CB, 145
UCT_EP_PARAM_FIELD_SOCKADDR_PACK_CB, 145
UCT_EP_PARAM_FIELD_USER_DATA, 144
uct_ep_params_field, 144
uct_ep_params_t, 138
uct_ep_pending_add, 156
uct_ep_pending_purge, 157
uct_error_handler_t, 140
UCT_EVENT_RECV, 142
UCT_EVENT_RECV_SIG, 142
UCT_EVENT_SEND_COMP, 142
UCT_FLUSH_FLAG_CANCEL, 143
UCT_FLUSH_FLAG_LOCAL, 143
uct_flush_flags, 142
uct_iface_addr_t, 138
uct_iface_attr_t, 136
uct_iface_close, 150
uct_iface_config_t, 135

© 2021 Unified Communication X (UCX). All rights reserved.
UCT_AM_TRACE_TYPE_RECV_DROP
    UCT Active messages, 180
UCT_AM_TRACE_TYPE_SEND
    UCT Active messages, 180
UCT_AM_TRACE_TYPE_SEND_DROP
    UCT Active messages, 180
uct_am_trace_type_t
    UCT Communication Resource, 137
uct_am_tracer_t
    UCT Active messages, 179
uct_async_event_cb_t
    UCT Communication Resource, 141
UCT_CB_FLAG_ASYNC
    UCT Communication Resource, 143
UCT_CB_FLAG_RESERVED
    UCT Communication Resource, 143
uct_cb_flags
    UCT Communication Resource, 143
UCT_CB_PARAM_FLAG_DESC
    UCT Communication Resource, 145
UCT_CB_PARAM_FLAG_FIRST
    UCT Communication Resource, 145
UCT_CB_PARAM_FLAG_MORE
    UCT Communication Resource, 145
uct_cb_param_flags
    UCT Communication Resource, 145
uct_cm_attr
    UCT Communication Resource, 138
uct_cm_attr_field
    UCT client-server operations, 204
UCT_CM_ATTR_FIELD_MAX_CONN_PRIV
    UCT client-server operations, 204
uct_cm_attr_t
    UCT Communication Resource, 136
uct_cm_client_ep_conn_notify
    UCT client-server operations, 209
uct_cm_close
    UCT client-server operations, 208
uct_cm_config_read
    UCT client-server operations, 208
uct_cm_config_t
    UCT Communication Resource, 136
uct_cm_ep_client_connect_args
    UCT client-server operations, 206
uct_cm_ep_client_connect_args_field
    UCT client-server operations, 206
UCT_CM_EP_CLIENT_CONNECT_ARGS_FIELD_REMOTE_DATA
    UCT client-server operations, 206
UCT_CM_EP_CLIENT_CONNECT_ARGS_FIELD_STATUS
    UCT client-server operations, 206
uct_cm_ep_client_connect_callback_t
    UCT client-server operations, 203
uct_cm_ep_client_connect_callback_t
    UCT client-server operations, 202
uct_cm_ep_server_conn_notify_args
    UCT client-server operations, 201
uct_cm_ep_server_conn_notify_args_field
    UCT client-server operations, 206
UCT_CM_EP_SERVER_CONN_NOTIFY_ARGS_FIELD_STATUS
    UCT client-server operations, 206
uct_cm_ep_server_conn_notify_args_t
    UCT client-server operations, 201
uct_cm_ep_server_conn_notify_callback_t
    UCT client-server operations, 202
uct_cm_h
    UCT Communication Resource, 138
uct_cm_listener_conn_request_args
    UCT client-server operations, 200
uct_cm_listener_conn_request_args_field
    UCT client-server operations, 205
UCT_CM_LISTENER_CONN_REQUEST_ARGS_CLIENT_ADDR
    UCT client-server operations, 206
UCT_CM_LISTENER_CONN_REQUEST_ARGS_CONN_REQUEST
    UCT client-server operations, 205
UCT_CM_LISTENER_CONN_REQUEST_ARGS_DEV_NAME
    UCT client-server operations, 205
UCT_CM_LISTENER_CONN_REQUEST_ARGS_REMOTE_DATA
    UCT client-server operations, 205
uct_cm_listener_conn_request_args_t
    UCT client-server operations, 201
uct_cm_listener_conn_request_callback_t
    UCT client-server operations, 202
uct_cm_open
    UCT client-server operations, 207
uct_cm_query
    UCT client-server operations, 208
uct_cm_remote_data
    UCT client-server operations, 204
uct_cm_remote_data_field
    UCT client-server operations, 205
UCT_CM_REMOTE_DATA_CONN_PRIV_DATA
    UCT client-server operations, 205
UCT_CM_REMOTE_DATA_CONN_PRIV_DATA_LENGTH
    UCT client-server operations, 205
UCT_CM_REMOTE_DATA_DEV_ADDR
    UCT client-server operations, 205
UCT_CM_REMOTE_DATA_DEV_ADDR_LENGTH
    UCT client-server operations, 205
uct_cm_remote_data_t
    UCT client-server operations, 201
uct_cm_t
    UCT Communication Resource, 138
uct_completion
    UCT Communication Resource, 133
uct_completion_callback_t
    UCT Communication Resource, 139
uct_completion_t
    UCT Communication Resource, 137
uct_completion_update_status
    UCT Communication Resource, 159
uct_component_attr
    UCT Communication Resource, 126
uct_component_attr_field
    UCT Communication Resource, 141
UCT_IFACE_FLAG_EVENT_SEND_COMP
UCT interface for asynchronous event capabilities, 217

UCT_IFACE_FLAG_GET_BCOPY
UCT interface operations and capabilities, 214

UCT_IFACE_FLAG_GET_SHORT
UCT interface operations and capabilities, 213

UCT_IFACE_FLAG_GET_ZCOPY
UCT interface operations and capabilities, 214

UCT_IFACE_FLAG_PENDING
UCT interface operations and capabilities, 213

UCT_IFACE_FLAG_PUT_BCOPY
UCT interface operations and capabilities, 213

UCT_IFACE_FLAG_PUT_SHORT
UCT interface operations and capabilities, 213

UCT_IFACE_FLAG_PUT_ZCOPY
UCT interface operations and capabilities, 213

UCT_IFACE_FLAG_TAG_EAGER_BCOPY
UCT interface operations and capabilities, 216

UCT_IFACE_FLAG_TAG_EAGER_SHORT
UCT interface operations and capabilities, 216

UCT_IFACE_FLAG_TAG_EAGER_ZCOPY
UCT interface operations and capabilities, 216

UCT_IFACE_FLAG_TAG_RNDV_ZCOPY
UCT interface operations and capabilities, 216

uct_iface_flush
UCT Communication Resource, 155

uct_iface_get_address
UCT Communication Resource, 151

uct_iface_get_device_address
UCT Communication Resource, 150

uct_iface_h
UCT Communication Resource, 135

uct_iface_is_reachable
UCT Communication Resource, 151

uct_iface_mem_alloc
UCT Communication Resource, 153

uct_iface_mem_free
UCT Communication Resource, 153

uct_iface_open
UCT Communication Resource, 149

uct_iface_open_mode
UCT Communication Resource, 143

UCT_IFACE_OPEN_MODE_DEVICE
UCT Communication Resource, 143

UCT_IFACE_OPEN_MODE_SOCKADDR_CLIENT
UCT Communication Resource, 144

UCT_IFACE_OPEN_MODE_SOCKADDR_SERVER
UCT Communication Resource, 144

UCT_IFACE_PARAM_FIELD_ASYNC_EVENT_ARG
UCT Communication Resource, 144

UCT_IFACE_PARAM_FIELD_ASYNC_EVENT_CB
UCT Communication Resource, 144

UCT_IFACE_PARAM_FIELD_CPU_MASK
UCT Communication Resource, 144

UCT_IFACE_PARAM_FIELD_DEVICE
UCT Communication Resource, 144

UCT_IFACE_PARAM_FIELD_ERR_HANDLER
UCT Communication Resource, 144

UCT_IFACE_PARAM_FIELD_ERR_HANDLER_ARG
UCT Communication Resource, 144

UCT_IFACE_PARAM_FIELD_ERR_HANDLER_FLAGS
UCT Communication Resource, 144

UCT_IFACE_PARAM_FIELD_ERR_HANDLER_ARG
UCT Communication Resource, 144

UCT_IFACE_PARAM_FIELD_ERR_HANDLER_FLAGS
UCT Communication Resource, 144

UCT_IFACE_PARAM_FIELD_HW_TM_EAGER_ARG
UCT Communication Resource, 144

UCT_IFACE_PARAM_FIELD_HW_TM_EAGER_CB
UCT Communication Resource, 144

UCT_IFACE_PARAM_FIELD_HW_TM_RNDV_ARG
UCT Communication Resource, 144

UCT_IFACE_PARAM_FIELD_HW_TM_RNDV_CB
UCT Communication Resource, 144

UCT_IFACE_PARAM_FIELD_KEEPALIVE_INTERVAL
UCT Communication Resource, 144

UCT_IFACE_PARAM_FIELD_OPEN_MODE
UCT Communication Resource, 144

UCT_IFACE_PARAM_FIELD_RX_HEADROOM
UCT Communication Resource, 144

UCT_IFACE_PARAM_FIELD_SOCKADDR
UCT Communication Resource, 144

uct_iface_params, 130

uct_iface_params.mode, 131

uct_iface_params.mode.device, 131

uct_iface_params.mode.sockaddr, 131

uct_iface_params_field

uct_iface_params_t
UCT Communication Resource, 136

uct_iface_query
UCT Communication Resource, 144

uct_iface_progress_disable
UCT Communication Resource, 159

uct_iface_progress_enable
UCT Communication Resource, 158

uct_iface_query
UCT Communication Resource, 150

uct_iface_reject
UCT Communication Resource, 144

uct_iface_reject
UCT client-server operations, 207

uct_iface_release_desc
UCT Active messages, 181

uct_iface_set_am_handler
UCT Active messages, 180

uct_iface_set_am_tracer
UCT Active messages, 180

uct_iface_tag_recv_cancel
UCT Tag matching operations, 194

uct_iface_tag_recv_zcopy
UCT Tag matching operations, 194

uct_iov
134

uct_iov_t

UCT Communication Resource, 139

uct_listener_attr
199

uct_listener_attr_field
UCT client-server operations, 204

UCT_LISTENER_ATTR_FIELD_SOCKADDR
uct_rkey_t
   UCT Communication Resource, 136
uct_rkey_unpack
   UCT Memory Domain, 176
UCT_SEND_FLAG_SIGNALED
   UCT Active messages, 179
UCT_SOCKET_ADDR_ACC_LOCAL
   UCT Memory Domain, 170
UCT_SOCKET_ADDR_ACC_REMOTE
   UCT Memory Domain, 170
uct_sockaddr_accessibility_t
   UCT Memory Domain, 169
uct_sockaddr_conn_request_callback_t
   UCT client-server operations, 201
uct_tag_context
   UCT Communication Resource, 138
UCT_TAG_RECV_CB_INLINE_DATA
   UCT Communication Resource, 145
uct_tag_t
   UCT Communication Resource, 138
uct_tag_unexp_eager_cb_t
   UCT Tag matching operations, 188
uct_tag_unexp_rndv_cb_t
   UCT Tag matching operations, 189
uct_tl_resource_desc
   UCT Communication Resource, 135
uct_tl_resource_desc_t
   UCT Communication Resource, 135
uct_unpack_callback_t
   UCT Communication Resource, 141
uct_worker_cb_id_t
   UCT Communication Resource, 139
uct_worker_create
   UCT Communication Context, 161
uct_worker_destroy
   UCT Communication Context, 161
uct_worker_h
   UCT Communication Resource, 137
uct_worker_progress
   UCT Communication Context, 163
uct_worker_progress_register_safe
   UCT Communication Context, 161
uct_worker_progress_unregister_safe
   UCT Communication Context, 162
Unified Communication Protocol (UCP) API, 11
Unified Communication Services (UCS) API, 218
Unified Communication Transport (UCT) API, 121
unpack
   UCT Data type routines, 120