Qualcomm Technologies, Inc.

Qualcomm Intelligent Multimedia Software Development Kit (IM SDK) Reference

80-70018-50 Rev. AD

May 2, 2025

© Qualcomm Technologies, Inc. and/or its subsidiaries. All rights reserved.

Contents

1 Overview of Qualcomm IM SDK 19

2 Sample applications 22

2.1 Run AI/ML sample applications 26

2.1.1 Download model and label files 27

2.1.2 Image classification 28

2.1.3 Object detection 33

2.1.4 Pose detection 38

2.1.5 Image segmentation 42

2.1.6 Image segmentation using Python with container 46

2.1.7 Parallel inferencing 50

2.1.8 Multi input/output object detection 55

2.1.9 Daisy chain detection and classification 59

2.1.10 Daisy chain detection and pose estimation 65

2.1.11 Monodepth from video 68

2.1.12 Video super-resolution 71

2.1.13 Multistream inference 75

2.1.14 Multistream batch inference 80

2.1.15 AI smart codec 85

2.1.16 Face detection 86

2.1.17 Face recognition 89

2.1.18 Audio classification 96

2.1.19 Metadata parsing 99

2.1.20 AI USB camera 103

2.2 Run multimedia sample applications 109

2.2.1 Run sample applications for camera 110

2.2.2 Run sample applications for video and audio 144

3 Python applications 173

3.1 Prerequisites 174

3.2 Camera encoding 175

3.3 Camera streaming using OpenCV . . . 176
3.4 Video transformation using OpenCV . . . 178
3.5 Concurrent video playback (video wall) using Python . . . 180
3.6 Multi-camera streaming using Python . . . 181
3.7 Object detection and display . . . 183
3.8 Decode and object detection using RTSP stream . . . 186
3.9 Decode JPEG images using Python . . . 189
3.10 Object detection and classification . . . 190
3.11 Transform and encode a camera stream . . . 193
3.12 Camera encode, object detection, and display . . . 194
3.13 Object detection, classification, and segmentation . . . 196
3.14 Parallel inference using Python . . . 199
3.15 Daisy chain detection and pose detection using Python . . . 202
3.16 Customize a Python sample application . . . 205

4 Deploy Qualcomm IM SDK as a container . . . 206
4.1 Build Qualcomm IM SDK docker image . . . 207
4.2 Load and run the docker . . . 209
4.3 Run docker image for multimedia/AI use cases . . . 210
4.4 Run container use cases for camera . . . 211

5 GStreamer command-line use cases . . . 212
5.1 Run machine learning use cases . . . 212
5.1.1 LiteRT use cases . . . 213
5.1.2 Qualcomm Neural Processing SDK use cases . . . 247
5.2 Run multimedia use cases . . . 267
5.2.1 Camera . . . 267
5.2.2 Video encode and decode . . . 271
5.2.3 Multi-camera/Multi-client use cases . . . 297
5.2.4 Transform and Transcode use cases . . . 303
5.2.5 Video playback use cases . . . 313
5.2.6 Audio use cases . . . 316
5.2.7 Audio-video use cases . . . 318
5.2.8 Video composition use cases . . . 320
5.2.9 Video overlay use cases . . . 323
5.2.10 Video split use cases . . . 329

6 Qualcomm GStreamer architecture and plugins . . . 331
6.1 GStreamer plugin architecture . . . 332
6.1.1 Graphics and display architecture . . . 333

6.1.2 Camera architecture 334
6.1.3 Video architecture 335
6.1.4 Audio architecture 337
6.1.5 Machine learning architecture 341
6.2 Configure Qualcomm GStreamer plugins 343
6.2.1 Configure display, camera, encode and decode plugins 343
6.2.2 Configure audio plugins 389
6.2.3 Configure ML plugins 397
6.3 Develop plugins using Qualcomm IM SDK 431
6.3.1 Customize a plugin 431
6.3.2 Create a plugin 434
7 Use GStreamer utilities for custom plugins 438
7.1 GSt APIs 438
7.1.1 Video APIs 439
7.1.2 Machine learning APIs 443
7.2 GSt debug 451
7.3 GStD framework 451
7.3.1 Run local use cases 453
7.3.2 Run TCP use cases 455
7.3.3 Run HTTP use cases 457
7.3.4 Enable message support through MQTT 460
8 Appendix: Qualcomm IM SDK plugin sizes 464

Figures

Figure 1-1: Qualcomm IM SDK overview 20
Figure 2-1: gst-ai-classification pipeline 29
Figure 2-2: Expected output for gst-ai-classification application 31
Figure 2-3: gst-ai-object-detection pipeline 33
Figure 2-4: Expected output for gst-ai-object-detection application 36
Figure 2-5: gst-ai-pose-detection pipeline 38
Figure 2-6: Expected output for gst-ai-pose-detection application 40
Figure 2-7: gst-ai-segmentation pipeline 42
Figure 2-8: Expected output for gst-ai-segmentation application 44
Figure 2-9: Workflow to set up host container 46
Figure 2-10: Expected output for image segmentation using Python binding 50
Figure 2-11: gst-ai-parallel-inference pipeline 51
Figure 2-12: Expected output for gst-ai-parallel-inference application 53
Figure 2-13: Multi-input inferencing pipeline 55
Figure 2-14: Expected output for gst-ai-multi-input-output-object-detection application–Preview 58
Figure 2-15: gst-ai-daisychain-detection-classification source pipeline 59
Figure 2-16: gst-ai-daisychain-detection-classification inference pipeline 60
Figure 2-17: Expected output for gst-ai-daisychain-detection-classification application 62
Figure 2-18: gst-ai-daisychain-detection-pose pipeline 65
Figure 2-19: Expected output for gst-ai-daisychain-detection-pose application 66
Figure 2-20: gst-ai-monodepth pipeline 68
Figure 2-21: Expected output for gst-ai-monodepth application 70
Figure 2-22: gst-ai-superresolution pipeline (Wayland display) 72
Figure 2-23: gst-ai-superresolution pipeline (file sink) 72

Figure 2-24: Expected output for VSR.....74
Figure 2-25: gst-ai-multistream-inference pipeline.....75
Figure 2-26: Expected output for gst-ai-multistream-inference.....78
Figure 2-27: gst-ai-multistream-batch-inference pipeline.....80
Figure 2-28: Expected output for gst-ai-multistream-batch-inference–Preview.....83
Figure 2-29: gst-ai-face-detection pipeline.....86
Figure 2-30: gst-ai-face-recognition pipeline.....90
Figure 2-31: gst-ai-audio-classification pipeline.....96
Figure 2-32: gst-ai-metadata-parser-example pipeline.....99
Figure 2-33: Expected output for gst-ai-metadata-parser-example application–object detection.....101
Figure 2-34: gst-ai-usb-single-camera-app pipeline.....103
Figure 2-35: gst-ai-usb-single-camera-app pipeline with object detection.....104
Figure 2-36: Expected output for gst-ai-usb-camera-app application–Preview.....107
Figure 2-37: Expected output for gst-ai-usb-camera-app application–object detection.....107
Figure 2-38: gst-camera-single-stream-example pipeline.....110
Figure 2-39: Expected output for gst-camera-single-stream-example application.....112
Figure 2-40: gst-usb-single-camera-app pipeline.....113
Figure 2-41: Expected output for gst-usb-single-camera-app application - Preview.....114
Figure 2-42: gst-multi-camera-example pipeline.....116
Figure 2-43: Expected output for gst-multi-camera-example application–Preview.....117
Figure 2-44: gst-multi-stream-example pipeline.....118
Figure 2-45: Expected output for gst-multi-stream-example application - Preview.....119
Figure 2-46: gst-snapshot-stream-example pipeline.....120
Figure 2-47: gst-activate-deactivate-streams-runtime-example pipeline.....122
Figure 2-48: Expected output for gst-activate-deactivate-streams-runtime-example application.....124
Figure 2-49: gst-add-remove-streams-runtime pipeline.....125
Figure 2-50: Expected output for gst-add-remove-streams-runtime application–Preview.....126
Figure 2-51: gst-appsink-example pipeline.....127
Figure 2-52: Expected output for gst-appsink-example application.....128
Figure 2-53: gst-camera-metadata-example pipeline.....129
Figure 2-54: Expected output for gst-camera-metadata-example application.....131

Figure 2-55: gst-add-streams-as-bundle-example pipeline..... 132
Figure 2-56: Expected output for gst-add-streams-as-bundle-example application.....133
Figure 2-57: gst-camera-burst-capture-example pipeline.....134
Figure 2-58: gst-camera-switch-example pipeline..... 136
Figure 2-59: Expected output for gst-camera-switch-example application..... 137
Figure 2-60: Output without sHDR..... 140
Figure 2-61: Output with sHDR..... 140
Figure 2-62: Output without LDC.....141
Figure 2-63: Output with LDC..... 141
Figure 2-64: gst-jpg-decode-example pipeline.....142
Figure 2-65: gst-concurrent-videoplay-composition pipeline with 4 streams.....144
Figure 2-66: Expected output for gst-concurrent-videoplay-composition application..... 147
Figure 2-67: gst-smartcodec-example pipeline for two streams.....148
Figure 2-68: gst-videocodec-concurrent-playback pipeline..... 149
Figure 2-69: gst-audio-video-playback pipeline..... 152
Figure 2-70: Expected output for gst-audio-video-playback application..... 154
Figure 2-71: gst-video-transcode-example pipeline.....155
Figure 2-72: gst-video-playback-example pipeline.....157
Figure 2-73: Expected output for gst-video-playback-example application.....159
Figure 2-74: gst-audio-video-encode pipeline..... 160
Figure 2-75: gst-audio-decode-example pipeline..... 162
Figure 2-76: gst-audio-encode-example pipeline..... 164
Figure 2-77: gst-transform-example pipeline.....166
Figure 2-78: Expected output for gst-transform-example application–Preview..... 169
Figure 2-79: gst-weston-composition-example pipeline.....170
Figure 2-80: Expected output for gst-weston-composition-example application.....172
Figure 3-1: Pipeline for camera encoding..... 175
Figure 3-2: gst-camera-opencv-resize.py pipeline..... 176
Figure 3-3: Expected output for gst-camera-opencv-resize.py..... 177
Figure 3-4: gst-opencv-transform.py pipeline.....178
Figure 3-5: Expected output for gst-opencv-transform.py application–Preview..... 179

Figure 3-6: gst-concurrent-videoplay-composition.py pipeline with 4 streams....180
Figure 3-7: Expected output for gst-concurrent-videoplay-composition.py application.... 180
Figure 3-8: gst-multi-camera-example.py pipeline.... 181
Figure 3-9: Expected output for gst-multi-camera-example.py application–Preview....182
Figure 3-10: Pipeline for object detection and preview....183
Figure 3-11: Expected output for object detection and display application–Preview.... 184
Figure 3-12: Pipeline for decode and object detection using RTSP stream–Preview....186
Figure 3-13: Expected output for object detection and display using RTSP stream–Preview....187
Figure 3-14: gst-jpg-image-decode.py pipeline....189
Figure 3-15: Expected output for gst-jpg-image-decode.py application–Preview....189
Figure 3-16: Pipeline for object detection and classification.... 190
Figure 3-17: Pipeline for video transformation....193
Figure 3-18: Pipeline for camera encode and object detection.... 194
Figure 3-19: Pipeline for object detection, image classification, and segmentation.... 196
Figure 3-20: gst-parallel-inference.py pipeline.... 199
Figure 3-21: Expected output for gst-ai-parallel-inference.py application.... 201
Figure 3-22: gst-ai-daisychain-detection-pose.py pipeline.... 202
Figure 3-23: Expected output for gst-ai-daisychain-detection-pose.py application.... 203
Figure 4-1: Qualcomm IM SDK within a container - architecture.... 206
Figure 5-1: Pipeline for classification overlay....214
Figure 5-2: Pipeline for classification with qtivcomposer.... 216
Figure 5-3: Pipeline for classification overlay and encode....218
Figure 5-4: Pipeline for classification and encode with qtivcomposer....220
Figure 5-5: Pipeline flow for audio classification and display.... 222
Figure 5-6: Pipeline flow for audio classification and decode–FLAC decoder....223
Figure 5-7: Pipeline flow for audio classification and decode–mpg123audiodec decoder....223
Figure 5-8: Pipeline for bounding box overlay.... 225
Figure 5-9: Pipeline for bounding box mask with qtivcomposer.... 227
Figure 5-10: Pipeline for bounding box overlay and encode.... 229
Figure 5-11: Pipeline for bounding box mask and encode with qtivcomposer....231
Figure 5-12: Pipeline for segmentation with qtivcomposer....233

Figure 5-13: Pipeline for segmentation and encode with qtivcomposer.....235
Figure 5-14: Pipeline for pose estimation overlay.....237
Figure 5-15: Pipeline for pose estimation mask using qtivcomposer.....239
Figure 5-16: Pipeline for pose estimation and encode using qtioverlay.....241
Figure 5-17: Pipeline for pose estimation and encode using qtivcomposer.....243
Figure 5-18: Pipeline for video resolution (Wayland display).....245
Figure 5-19: Pipeline for camera to RTSP.....245
Figure 5-20: Pipeline for camera to RTSP with ML detection.....246
Figure 5-21: Preview with RTSP source.....246
Figure 5-22: Pipeline for classification overlay.....247
Figure 5-23: Pipeline for classification using qtivcomposer.....249
Figure 5-24: Pipeline for classification overlay and encode.....251
Figure 5-25: Pipeline for classification and encode with qtivcomposer.....253
Figure 5-26: Pipeline for bounding box overlay.....255
Figure 5-27: Pipeline for bounding box mask with qtivcomposer.....257
Figure 5-28: Pipeline for bounding box overlay and encode.....259
Figure 5-29: Pipeline for bounding box mask and encode with qtivcomposer.....261
Figure 5-30: Pipeline for segmentation with qtivcomposer.....263
Figure 5-31: Pipeline for image segmentation and encode with qtivcomposer.....265
Figure 5-32: 1080p camera stream sent to socket, read from socket, encoded to AVC and stored in file.....271
Figure 5-33: Pipeline for encoding 1080p RTSP stream.....271
Figure 5-34: Pipeline for encoding two (4k and 1080p) camera streams.....273
Figure 5-35: Pipeline for encoding three 1080p camera streams.....274
Figure 5-36: Pipeline for encoding 720p AVC, 720p HEVC, and 720p YUV camera streams.....275
Figure 5-37: Pipeline for encoding 720p AVC MP4, 720p AVC MPEGTS, and 720p AVC MP4 camera streams.....276
Figure 5-38: Pipeline for encoding three 1080p camera streams.....277
Figure 5-39: Pipeline for three camera streams–4k to JPEG snapshot, 1080p to encoding, 1080 to display.....278
Figure 5-40: Pipeline for two camera clients–encoding and preview.....299
Figure 5-41: Two streams from primary and secondary cameras–side by side display.....300
Figure 5-42: Pipeline for two 720p camera streams–side by side display, encoded, and streamed through RTSP.....301

Figure 5-43: Pipeline for two 720p streams–picture-in-picture display....301
Figure 5-44: Pipeline for two 720p camera streams–picture-in-picture display, encoded, and streamed through RTSP.... 303
Figure 5-45: Pipeline for rotation and display–camera source.... 304
Figure 5-46: Pipeline for rotation and display–file source.... 304
Figure 5-47: Pipeline for 4k image downscaled to 1080p, rotated, and displayed–camera source....305
Figure 5-48: Pipeline for 4k stream downscaled to 1080p, and, rotated, and displayed–file source.... 306
Figure 5-49: Pipeline for displaying flipped images–camera source.... 306
Figure 5-50: Pipeline for displaying flipped images–file source.... 307
Figure 5-51: Pipeline for displaying flipped and downscaled images–camera source.... 308
Figure 5-52: Pipeline for 4k video–encoding and downscaling to 1080p.... 309
Figure 5-53: Pipeline to decode and display a single stream (H.264).... 313
Figure 5-54: Pipeline to decode and save a stream to a file (H.264).... 313
Figure 5-55: Pipeline to decode and display a single stream (H.265).... 314
Figure 5-56: Pipeline to decode and save a stream to a file (H.265).... 314
Figure 5-57: Pipeline to decode and display a single stream (VP9).... 314
Figure 5-58: Pipeline to decode and save a stream to a file (VP9)....314
Figure 5-59: Simultaneous playback of two video streams from a file.... 315
Figure 5-60: Pipeline for audio capture with PCM encode.... 316
Figure 5-61: Pipeline for the audio capture with FLAC encode....316
Figure 5-62: Pipeline for WAV stream playback.... 317
Figure 5-63: Pipeline for MP2/MP3 stream playback.... 317
Figure 5-64: Pipeline for FLAC stream playback....317
Figure 5-65: Pipeline for OGG stream playback.... 318
Figure 5-66: Pipeline for AV capture–camera source and audio source.... 319
Figure 5-67: Pipeline for AV playback–AVC video + MP3 audio.... 319
Figure 5-68: Pipeline for AV playback–AVC video + FLAC audio....320
Figure 5-69: Pipeline for side by side display of 16 480p video streams–file sources.... 322
Figure 5-70: Pipeline for picture-in-picture display of 1080p streams–file source.... 322
Figure 5-71: Pipeline for picture-in-picture display of 1080p streams–file source and camera source....323
Figure 5-72: Pipeline for text overlay on video–file source.... 324

Figure 5-73: Pipeline for text overlay on video–camera source....324
Figure 5-74: Pipeline for text date and time overlay on video–file source.... 324
Figure 5-75: Pipeline for text date and time overlay on video–camera source.... 325
Figure 5-76: Pipeline for static image overlay on video–file source.... 325
Figure 5-77: Pipeline for static image overlay on video–camera source....326
Figure 5-78: Pipeline for bounding box overlay on video–file source....326
Figure 5-79: Pipeline for bounding box overlay on video–camera source....326
Figure 5-80: Pipeline for inverse privacy mask overlay on video–file source....327
Figure 5-81: Pipeline for inverse privacy mask overlay on video–camera source....327
Figure 5-82: Pipeline for non-inverse privacy mask overlay on video–file source.... 328
Figure 5-83: Pipeline for non-inverse privacy mask overlay on video–camera source.... 328
Figure 5-84: Pipeline for qtivsplit composing 9 1080p streams from one input video file.... 330
Figure 6-1: Qualcomm IM SDK GStreamer plugin architecture.... 332
Figure 6-2: Weston/Wayland architecture....333
Figure 6-3: Camera pipeline....334
Figure 6-4: Video encode pipeline.... 336
Figure 6-5: Video decode pipeline.... 337
Figure 6-6: Audio capture pipeline.... 338
Figure 6-7: Audio playback pipeline....339
Figure 6-8: Audio encoding pipeline.... 340
Figure 6-9: Audio decode pipeline.... 340
Figure 6-10: ML pipeline.... 342
Figure 6-11: Pipeline with qtivcomposer.... 353
Figure 6-12: GStreamer pipeline with qtivsplit.... 361
Figure 6-13: qtivsplit buffer management.... 361
Figure 6-14: Gstbuffer workflow with qtimlvconverter.... 398
Figure 6-15: Gstbuffer workflow with qtibatch....401
Figure 6-16: Tensor mode qtimlsnpe architecture....403
Figure 6-17: GstBuffer workflow with qtimltflite.... 405
Figure 6-18: Postprocessing for object detection architecture....409
Figure 6-19: qtimlvclassification in pipeline....409

Figure 6-20: GstBuffer workflow with qtimlvdetection 412
Figure 6-21: Postprocessing for object detection architecture 413
Figure 6-22: GstBuffer workflow with qtimlvsegmentation 415
Figure 6-23: Postprocessing for pose estimation architecture 417
Figure 6-24: GstBuffer workflow with qtimlvpose 418
Figure 6-25: GstBuffer workflow for qtimlvsuperresolution 419
Figure 6-26: GStreamer pipeline with qtimetamux 424
Figure 6-27: qtimldemux architecture 426
Figure 6-28: qtismartvencbin architecture 429
Figure 7-1: GstImageBufferPool workflow 439
Figure 7-2: Hardware-accelerated image manipulation, transformation, and color conversion 441
Figure 7-3: GstBufferPool subclass for ML 450

Tables

Table 2-1: Sample model and label files for gst-ai-classification....29
Table 2-2: Sample model and label files for gst-ai-object-detection....34
Table 2-3: Field description–config_detection.json file....37
Table 2-4: Sample model and label files for gst-ai-pose-detection....38
Table 2-5: Field description–config_pose.json file....41
Table 2-6: Sample model and label files for gst-ai-segmentation....42
Table 2-7: Field description–config_segmentation.json file....45
Table 2-8: Sample model and label files for gst-ai-parallel-inference....51
Table 2-9: Sample model and label files for gst-ai-multi-input-output-object-detection....55
Table 2-10: Sample model and label files for gst-ai-daisychain-detection-classification....60
Table 2-11: Field description–config_daisychain_detection_classification.json file....64
Table 2-12: Sample model and label files for gst-ai-daisychain-detection-pose....65
Table 2-13: Sample model and label files for gst-ai-monodepth....68
Table 2-14: Field description–config_monodepth.json file....71
Table 2-15: Sample model for gst-ai-superresolution....72
Table 2-16: Sample model and label files for gst-ai-multistream-inference....76
Table 2-17: Field description–gst-ai-multistream-inference file....79
Table 2-18: Sample model and label files for gst-ai-multistream-batch-inference....81
Table 2-19: Field description–gst-ai-multistream-batch-inference file....84
Table 2-20: Sample model and label files for gst-ai-smartcodec-example....85
Table 2-21: Sample model and label files for gst-ai-face-detection....87
Table 2-22: Field description–gst-ai-face-detection file....89
Table 2-23: Sample model and label files for gst-ai-face-detection....90
Table 2-24: Field description–config-face-recognition.json file....95

Table 2-25: Sample model and label files for gst-ai-audio-classification.... 97
Table 2-26: Sample model and label files for gst-ai-object-detection.... 100
Table 2-27: Field description–gst-ai-metadata-parser-example file.... 103
Table 2-28: Sample model and label files for gst-ai-usb-camera-app.... 104
Table 2-29: Expected output for gst-usb-single-camera-app.... 106
Table 2-30: Field description–config_detection.json file.... 108
Table 2-31: Expected output for gst-camera-single-stream-example.... 112
Table 2-32: Expected output for gst-ai-usb-single-camera-app–without AI.... 114
Table 2-33: Camera imaging features.... 142
Table 2-34: Smart codec–file sizes.... 148
Table 3-1: Python applications.... 173
Table 3-2: Sample model and label files for object detection and display python application.... 183
Table 3-3: Pipeline processing stages for object detection and preview.... 185
Table 3-4: Default directories for model and label files for object detection using RTSP stream.... 187
Table 3-5: Models used for detection and classification.... 190
Table 3-6: Default directories for model and label files for object detection and classification python application.... 191
Table 3-7: Pipeline processing stages for object detection and classification.... 191
Table 3-8: Default model and label files for gst-camera-two-stream-encode-file-detection-display.py.... 194
Table 3-9: Pipeline processing stages for camera encode and object detection.... 195
Table 3-10: Models used for detection and classification.... 196
Table 3-11: Default directories for model and label files.... 197
Table 3-12: Pipeline processing stages for object detection, image classification, and segmentation.... 197
Table 3-13: Default model and label files for gst-parallel-inference.py.... 200
Table 3-14: Sample model and label files for gst-ai-daisychain-detection-pose.... 205
Table 4-1: Components for Qualcomm IM SDK docker container.... 206
Table 5-1: Pipeline processing stages for classification overlay.... 214
Table 5-2: Pipeline processing stages for classification with qtivcomposer.... 216
Table 5-3: Pipeline processing stages for classification overlay and encode.... 218
Table 5-4: Pipeline processing stages for classification and encode with qtivcomposer.... 220
Table 5-5: Pipeline processing stages for audio classification.... 222

Table 5-6: Pipeline processing stages for audio classification using FLAC and mpg123audiodec decoders..... 224
Table 5-7: Pipeline processing stages for bounding box overlay.....225
Table 5-8: Pipeline processing stages for bounding box mask with qtivcomposer..... 227
Table 5-9: Pipeline processing stages for bounding box overlay and encode..... 229
Table 5-10: Pipeline processing stages for bounding box mask and encode with qtivcomposer..... 231
Table 5-11: Pipeline processing stages for segmentation with qtivcomposer..... 233
Table 5-12: Pipeline processing stages for segmentation and encode with qtivcomposer..... 235
Table 5-13: Pipeline processing stages for pose estimation overlay.....237
Table 5-14: Pipeline processing stages for pose estimation mask using qtivcomposer.....239
Table 5-15: Pipeline processing stages for pose estimation and encode using qtioverlay.....241
Table 5-16: Pipeline processing stages for pose estimation and encode using qtivcomposer.....243
Table 5-17: Pipeline processing stages for classification overlay..... 248
Table 5-18: Pipeline processing stages for classification using qtivcomposer..... 250
Table 5-19: Pipeline processing stages for classification overlay and encode..... 252
Table 5-20: Pipeline processing stages for classification and encode with qtivcomposer..... 254
Table 5-21: Pipeline processing stages for bounding box overlay.....256
Table 5-22: Pipeline processing stages for bounding box mask with qtivcomposer..... 257
Table 5-23: Pipeline processing stages for bounding box overlay and encode..... 260
Table 5-24: Pipeline processing stages for bounding box mask and encode with qtivcomposer..... 262
Table 5-25: Pipeline processing stages for segmentation with qtivcomposer..... 264
Table 5-26: Pipeline processing stages for image segmentation and encode with qtivcomposer.....266
Table 5-27: QCS8275 and QCS9075–GSML port support and limitation..... 298
Table 5-28: Multi camera client use cases for encoding and preview.....298
Table 6-1: Weston/Wayland components.....333
Table 6-2: Camera pipeline components..... 334
Table 6-3: Video encode components..... 336
Table 6-4: Video decode components..... 337
Table 6-5: Audio capture components.....338
Table 6-6: Audio playback components..... 339
Table 6-7: Audio encoding components..... 340
Table 6-8: Audio decode components..... 341

Table 6-9: ML pipeline processing stages....342
Table 6-10: Good, bad, ugly GSt plugins....343
Table 6-11: Pad templates for qtiqmmfsrc....344
Table 6-12: Pad properties of qtiqmmfsrc....345
Table 6-13: Element properties of qtiqmmfsrc....346
Table 6-14: Element signals for qtiqmmfsrc....352
Table 6-15: Pad templates for qtivcomposer....354
Table 6-16: Pad properties of qtivcomposer....355
Table 6-17: Element properties of qtivcomposer....356
Table 6-18: Pad templates for Waylandsink....357
Table 6-19: Element properties of Waylandsink....358
Table 6-20: Operational mode properties of qtivsplit....360
Table 6-21: Pad templates for qtivsplit....362
Table 6-22: Pad properties for qtivsplit....363
Table 6-23: Element properties of qtivsplit....363
Table 6-24: Pad templates for qtivtransform....365
Table 6-25: Elements properties of qtivtransform....365
Table 6-26: Configuration properties of qtisocketsink....367
Table 6-27: Pad templates for qtisocketsink....367
Table 6-28: Element properties for qtisocketsink....367
Table 6-29: Pad templates for qtisocketsrc....369
Table 6-30: Element properties of qtisocketsrc....369
Table 6-31: Pad templates for v4l2h264dec....370
Table 6-32: Element properties of v4l2h264dec....371
Table 6-33: Pad templates for v4l2h265dec....373
Table 6-34: Element properties of v4l2h265dec....373
Table 6-35: Pad templates for v4l2h264enc....375
Table 6-36: Element properties of v4l2h264enc....375
Table 6-37: v4l2ctl extra-controls–user controls....377
Table 6-38: v4l2ctl extra-controls–codec controls....377
Table 6-39: v4l2ctl extra-controls–H.264 codec controls....379

Table 6-40: Pad templates for v4l2h265enc....382
Table 6-41: Element properties of v4l2h265enc....383
Table 6-42: v4l2ctl–H.265 codec controls....384
Table 6-43: Pad templates for v4l2vp9dec....387
Table 6-44: Element properties of v4l2vp9dec....388
Table 6-45: Pad templates for pulsesrc....390
Table 6-46: Element properties of pulsesrc....390
Table 6-47: Pad templates for pulsesink....393
Table 6-48: Elements properties of pulsesink....394
Table 6-49: Channel transformation example....398
Table 6-50: Pad templates for qtimlvconverter....399
Table 6-51: Element properties for qtimlvconverter....399
Table 6-52: Pad templates for qtimlaconverter....400
Table 6-53: Element properties of qtimlaconverter....401
Table 6-54: Pad templates for qtibatch....402
Table 6-55: Element properties for qtibatch....402
Table 6-56: Pad templates for qtimlsnpe....403
Table 6-57: Element properties of qtimlsnpe....404
Table 6-58: Pad templates for qtimltflite....405
Table 6-59: Element properties for qtimltflite....405
Table 6-60: Pad template for qtimlqnn....407
Table 6-61: Element properties of qtimlqnn....407
Table 6-62: Pad templates for qtimlvclassification....409
Table 6-63: Element properties of qtimlvclassification....410
Table 6-64: Pad templates for qtimlaclassification....411
Table 6-65: Element properties of qtimlaclassification....411
Table 6-66: Pad templates for qtimlvdetection....414
Table 6-67: Element properties for qtimlvdetection....414
Table 6-68: Pad templates for qtimlvsegmentation....416
Table 6-69: Element properties for qtimlvsegmentation....416
Table 6-70: Pad templates for qtimlvpose....418

Table 6-71: Element properties of qtimlvpose....418
Table 6-72: Pad templates for qtimlvsuperresolution....420
Table 6-73: Element properties for qtimlvsuperresolution....420
Table 6-74: Pad templates for qtivoverlay.... 421
Table 6-75: Element properties for qtivoverlay.... 422
Table 6-76: Pad templates for qtimetamux....425
Table 6-77: Element properties of qtimetamux.... 425
Table 6-78: Pad template for qtimldemux.... 427
Table 6-79: qtimldemux element properties.... 427
Table 6-80: Pad templates for qtirtspbin.... 427
Table 6-81: Element properties of qtirtspbin....428
Table 6-82: Pad template for qtismartvencbin....429
Table 6-83: Element properties of qtismartvencbin.... 430
Table 7-1: GSt video and ML APIs.... 438
Table 7-2: Information structure describing ML properties....444
Table 7-3: GstMLTensorMeta information structure....447
Table 7-4: Structure of GstMLFrame....449

1 Overview of Qualcomm IM SDK

The Qualcomm® Intelligent Multimedia SDK (IM SDK) is a unified SDK across Internet of Things (IOT) segments. This SDK allows seamless multimedia and AI/ML application development.

Get started	Sample applications	GStreamer command-line use cases
→ Workflow → GStreamer plugin architecture	→ Sample applications → Run AI/ML sample applications → Run multimedia sample applications → Python applications	→ Run machine learning use cases → Run multimedia use cases

Qualcomm GStreamer plugins	GStreamer utilities
→ Configure ML plugins → Configure display, camera, encode and decode plugins → Configure audio plugins	→ GSt APIs → GSt debug → GStD framework

IMPORTANT

- Ensure that the host computer uses Ubuntu 22.04.
- The commands in this guide are compatible with Qualcomm® Linux® 1.4. If your release version isn't 1.4, verify your Qualcomm Linux release version and flash the latest version.
- The sample applications and AI procedures in this guide are compatible with Qualcomm AI® Runtime SDK v2.32 and LiteRT (or TFLite) v2.16.1. Ensure that you download the matching SDKs to your host computer before starting AI/ML development.
- See hardware SoCs that are supported on Qualcomm Linux. For information about all the supported SDKs, see SDK Overview.

The Qualcomm IM SDK provides the development environment with upstream and Qualcomm GStreamer plugins as APIs. You can use these APIs to develop and optimize applications, create pipelines, and customize plugins.

The figure shows a high-level overview of the Qualcomm IM SDK environment:

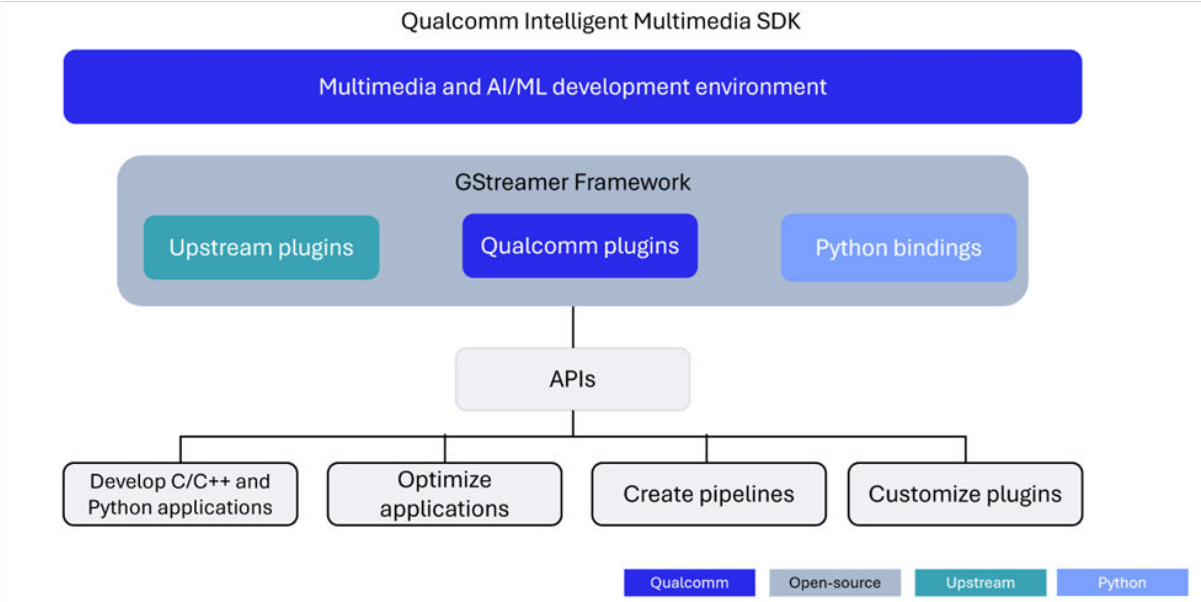


Figure 1-1 Qualcomm IM SDK overview

This video allows you to understand and use the Qualcomm IM SDK for creating applications on Qualcomm Linux. The video will open in a new tab.

See also

- Qualcomm Intelligent Multimedia Product (QIMP) SDK Quick Start Guide
- SDK Overview

2 Sample applications

The Qualcomm IM SDK includes sample applications that enable you to explore the **AI/ML** and **multimedia** features of Qualcomm Linux. Additionally, **Python-based samples** are available for common use cases. These applications function as a reference for running custom use cases on Qualcomm Linux.

IMPORTANT

- Ensure that the host computer uses Ubuntu 22.04.
- The commands in this guide are compatible with Qualcomm Linux 1.4. If your release version isn't 1.4, Verify your Qualcomm Linux release version and flash the latest version.
- The sample applications and AI procedures in this guide are compatible with Qualcomm AI Runtime SDK v2.32 and LiteRT (or TFLite) v2.16.1. Ensure that you download the matching SDKs to your host computer before starting AI/ML development.

Use the following sample applications to run different use cases related to retail, AI Edge Box, IP camera, drones, and robotics.

- **AI/ML applications** focus on the AI and ML capabilities.
- **Multimedia applications** use the camera, video, and audio features.
- **Python-based applications** use Python to run the AI/ML and multimedia use cases. For more information, see Python applications.

To run the AI/ML and multimedia sample applications using a GUI-based application without any host setup, see Run sample applications using Qdemo.

AI/ML sample applications

Application	CodeLinaro Git - source path	Description	QCS6490	QCS8275	QCS9075
Image classification	gst-ai-classification	Classification on streams from a camera, file source, or Real-Time Streaming Protocol (RTSP).	✔	✔	✔
Object detection	gst-ai-object-detection	Object detection on streams from a camera, file source, or RTSP.	✔	✔	✔
Pose detection	gst-ai-pose-detection	Pose detection on streams from a camera, file source, or RTSP.	✔	✔	✔

Application	CodeLinaro Git - source path	Description	QCS6490	QCS8275	QCS9075
Image segmentation	gst-ai-segmentation	Image segmentation on streams from a camera, file source, or RTSP.	✔	✔	✔
Parallel inferencing	gst-ai-parallel-inference	Many AI/ML models run in parallel on streams from a camera, file source, or RTSP.	✔	✖	✔
Multi input/ output object detection	gst-ai-multi-input-output-object-detection	Use cases for Edge AI boxes, which provide inputs through sources such as camera, file source, or RTSP.	✔	✔	✔
Daisy chain detection and classification	gst-ai-daisychain-detection-classification	Cascaded object detection and classification on images streamed from sources such as camera, file source, or RTSP.	✔	✔	✔
Daisy chain detection and pose estimation	gst-ai-daisychain-detection-pose	Cascaded object detection and pose detection from sources such as camera, file source, or RTSP.	✔	✔	✔
Monodepth from video	gst-ai-monodepth	Monodepth on images streamed from a camera, file source, or RTSP.	✔	✔	✔
Video super-resolution	gst-ai-superresolution	High resolution video frames from low-resolution input.	✔	✔	✖
Multistream inference	gst-ai-multistream-inference	AI inference (object detection and classification) from sources such as camera, file source, or RTSP.	✔	✔	✔
Multistream batch inference (Alpha release)	gst-ai-multistream-batch-inference	Batched AI inference (object detection and segmentation) from file source.	✔	✔	✔
AI smart codec	gst-ai-smartcodec-example	Reduce the network bandwidth or storage for input feed from camera or file source.	✔	✔	✔

Application	CodeLinaro Git - source path	Description	QCS6490	QCS8275	QCS9075
Face detection	gst-ai-face-detection	Takes inputs from a camera, file, or an RTSP stream and uses the Qualcomm® AI Hub detection model to produce a preview.	✔	✔	✔
Face recognition	gst-ai-face-recognition	Takes inputs from a camera or an RTSP stream for face recognition.	✔	✔	✔
Audio classification	gst-ai-audio-classification	Classification on streams from audio source.	✔	✔	✔
Metadata parsing	gst-ai-metadata-parser-example	Parses metadata using appsink plugin on streams from camera, file, or RTSP sources. Also, it provides the human count from the stream.	✔	✔	✔
AI USB camera	gst-ai-usb-camera-app	USB single camera streaming for preview, video encoder, or network (RTSP) along with object detection and preview	✔	✔	✔
Image segmentation using Python with container (Alpha release)	snpe_segmentation_app.py	Segmentation using python bindings from a docker.	✔	✔	✔

Multimedia sample applications

Application	CodeLinaro Git–source path	Description	QCS6490	QCS8275	QCS9075
Camera applications					
Single camera streaming	gst-camera-single-stream-example	Single camera streaming to preview, video encoder, or network (RTSP).	✔	✔	✔
Multi-camera streaming	gst-multi-camera-example	Concurrent streaming from two cameras. Use cases such as stereo and dash camera.	✔	✖	✖
Multistream	gst-multi-stream-example	Single camera data duplicated into two streams using tee.	✔	✖	✖

Application	CodeLinaro Git–source path	Description	QCS6490	QCS8275	QCS9075
Live snapshot	gst-snapshot-stream-example	Snapshot generation along with active video recording. Use cases such as using a surveillance camera that needs snapshots created based on events (such as intruder alerts).	✔	✖	✖
Activate-deactivate streams at runtime	gst-activate-deactivate-streams-runtime-example	Camera resolution modifications on-the-fly without sensor restart.	✔	✖	✖
Add-remove streams at runtime	gst-add-remove-streams-runtime	Addition and removal of many streams dynamically.	✔	✖	✖
Camera frame processing	gst-appsink-example	Application of transformations and other algorithms to a camera frame before encoding.	✔	✔	✔
Camera metadata	gst-camera-metadata-example	Metadata such as scene mode, white balance, and bit rate. Used for advanced imaging information in surveillance and drone cameras.	✔	✖	✖
Camera stream configuration	gst-add-streams-as-bundle-example	Configuration of camera streams as a bundle, thus saving on time.	✔	✖	✖
Camera burst capture	gst-camera-burst-capture-example	Snapshot capture in Burst mode.	✔	✖	✖
Camera switch	gst-camera-switch-example	Switch between two cameras.	✔	✖	✖
Wide dynamic range and electronic image stabilization	gst-camera-shdr-ldc-eis-example	Super high dynamic range (sHDR), Low Dynamic Range (LDR), and electronic image stabilization (EIS) camera imaging techniques.	✔	✖	✖
Decode JPEG images	gst-jpg-decode-example	Decode the JPEG images and preview them on a screen.	✔	✔	✔
Video applications					
Video wall	gst-concurrent-videoplay-composition	Play back the videos composed on a device. The use cases provide scenarios on using a video wall in retail spaces and digital signage.	✔	✔	✔

Application	CodeLinaro Git–source path	Description	QCS6490	QCS8275	QCS9075
Smart codec	gst-smartcodec-example	Reduce the network bandwidth or storage for a camera input.	✔	✔	✔
Multi format decode	gst-videocodec-concurrent-playback	Decoding and playback of many formats concurrently. Use cases for AI box and videoconferencing system, which sends video inputs in different formats.	✔	✔	✔
Audio-video playback	gst-audio-video-playback	Audio and video playback capabilities. Build of different multimedia products that require audio and video.	✔	✔	✔
Video transcoding	gst-video-transcode-example	Video transcoding of AVC-to-HEVC or HEVC-to-AVC format.	✔	✔	✔
Video playback	gst-video-playback-example	Decoding and playback of a video file.	✔	✖	✖
Video transformation	gst-transform-example	Transform a video stream (rotate, flip, and scale).	✔	✔	✔
Video composition	gst-weston-composition-example	Composition of various sources like a live camera input and an offline file.	✔	✔	✔
Audio and video encode	gst-audio-video-encode	Encoding of audio and video data, which is then stored into a user-specified output file.	✔	✔	✔
Audio applications					
Audio playback	gst-audio-decode-example	Playback of an audio file.	✔	✔	✔
Audio recording	gst-audio-encode-example	Records and encodes captured audio in FLAC or WAV formats.	✔	✔	✔

2.1 Run AI/ML sample applications

Use Qualcomm Linux AI/ML features with Qualcomm AI Runtime SDK (Qualcomm® Neural Processing SDK and Qualcomm AI Engine direct), and LiteRT (formerly TFLite) models.

2.1.1 Download model and label files

Download the model and label files for QCS6490, QCS9075, and QCS8275 to run the AI/ML sample applications.

Do the following on the Linux host computer:

1. Enable SSH and connect to Wi-Fi. For instructions, see Sign in using SSH.

   NOTE If SSH is already enabled and Wi-Fi is connected, skip this step.

2. Sign in to the target device using SSH:

   ```
   ssh root@<ip address of target device>
   ```

3. Download the automated script that allows you to download model and label files to the `etc/models` and `etc/labels` directories on the target device:

   ```
   curl -L -O https://raw.githubusercontent.com/quic/sample-apps-for-qualcomm-linux/refs/heads/main/download_artifacts.sh
   ```

   NOTE Run this script on the target device. Ensure that the target device has a valid internet connection.

4. Set permissions for the script:

   ```
   chmod +x download_artifacts.sh
   ```

5. Run the script with the required arguments to download model and label files on the target device:

   ```
   ./download_artifacts.sh -v GA1.4-rel -c <soc name>
   ```

   Here, replace `<soc name>` with either `QCS6490`, `QCS9075`, or `QCS8275`.

6. The YOLOv8 and YOLO-NAS models aren't available by default. You can use the following options to either download the models using a script or export them with AI Hub APIs.

   If you are using Multistream batch inference application, you can generate a batch model.

   - Download the models using a script:

     i. Create a Qualcomm AI Hub account.

     ii. Select the account name, then go to **Settings** in the upper right corner, and select the API token.

     iii. Export the models on the Linux host computer and set the required permissions:

     ```
     curl -L -O https://raw.githubusercontent.com/quic/sample-apps-for-qualcomm-linux/refs/heads/main/scripts/export_model.sh
     ```

     ```
     chmod +x export_model.sh
     ```

     Replace *API_TOKEN* with the selected key:

     ```
     ./export_model.sh --api-token=<API_TOKEN>
     ```

   - Export the models using AI Hub APIs:

     – YOLOv8-Detection-Quantized

     – Yolo-NAS-Quantized

     The current release (GA1.4) uses Qualcomm AI Runtime SDK v2.32.

For example, to export the YOLOv8 QNN model, run the following command:

```
python -m qai_hub_models.models.yolov8_det.export --quantize w8a8 --
target-runtime=qnn --chipset="qualcomm-qcs6490-proxy" --compile-
options="--qairt_version 2.32" --profile-options "--qairt_version 2.32"
```

For example, to export the YOLOv8 LiteRT model, run the following command:

```
python -m qai_hub_models.models.yolov8_det.export --quantize w8a8 --
target-runtime=tflite --chipset="qualcomm-qcs6490-proxy"
```

- Generate a batch model.

  To change the batch size of the model, update `<N>` in the following `export` command:

  ```
  python -m qai_hub_models.models.<Model_Name>.export --batch-size <N> --
  device "QCS6490 (Proxy)"
  ```

  For example, to export the YOLOv8 LiteRT model with`--batch-size 4`, run the following command:

  ```
  python -m qai_hub_models.models.yolov8_det.export --quantize w8a8 --
  target-runtime=tflite --chipset="qualcomm-qcs6490-proxy" --batch-size 4
  ```

7. Update the `q_offset` and `q_scale` constants of the quantized LiteRT model in the JSON file. For instructions, see Obtain model constants.

If any model isn't available after downloading the script file, you can download the model from IoT–Qualcomm AI Hub and push it on the target device:

```
scp <model filename> root@<IP addr of the target device>:/etc/models
```

For example:

```
scp mobilenet_v2_quantized.tflite root@<IP addr of the target device>:/etc/
models
```

NOTE If you want to run the sample applications from the UART shell, remount the file system with read/write permission using the following command on the target device:

```
mount -o remount,rw /usr
```

2.1.2 Image classification

The **gst-ai-classification** application allows you to identify the subject in an image. The use cases use the Qualcomm Neural Processing SDK, LiteRT, or Qualcomm AI Engine direct models.

The figure shows the pipeline, which receives a video stream from a camera, file source, or Real-Time Streaming Protocol (RTSP), does the preprocessing, runs the inference on the AI hardware, and displays the results.

For information about the plugins used for classification, see Pipeline flow.

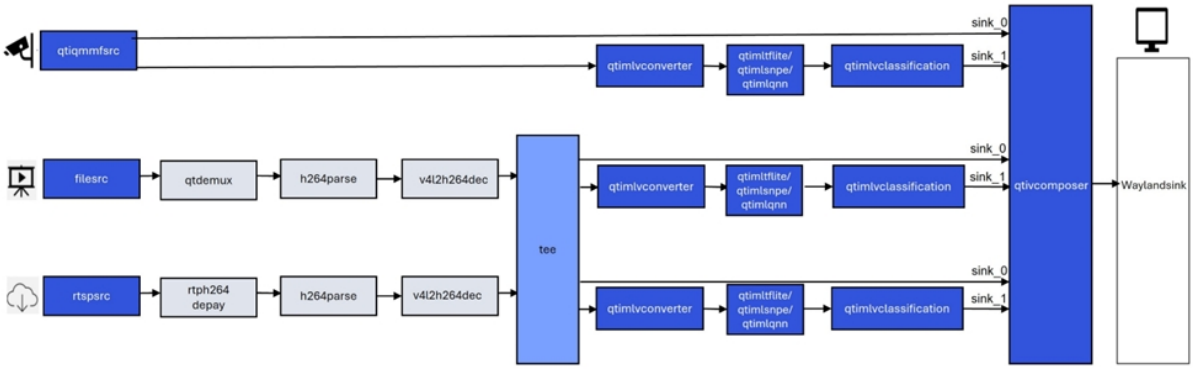


Figure 2-1 gst-ai-classification pipeline

Sample model and label files

Table 2-1 Sample model and label files for gst-ai-classification

Runtime	Model files	Label files
Qualcomm Neural Processing SDK	*inceptionv3.dlc*	*classification.labels*
LiteRT	*inception_v3_quantized.tflite*	
Qualcomm AI Engine direct	*inception_v3_quantized.bin*	

Prerequisites

- If not already done so, Download and install eSDK.
- To run the application, push the model and label files to the device. For instructions, see Download model and label files.

  The application supports the Qualcomm Neural Processing SDK, Qualcomm AI Engine direct, and LiteRT models.
- To access your host computer, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is already enabled, you can skip this step.
- Push the model files from the Linux host computer:

  ```
  scp <model_filename> root@<IP address of target device>:/etc/models
  ```
- Note that the Download model and label files downloads the sample `video.mp4` video to the `/etc/media` directory. If you are using a custom video, then ensure that you push the video to `/etc/media` and update the file path in the application config.JSON file.
- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.
- Enable the display:

  ```
  export XDG_RUNTIME_DIR=/dev/socket/weston && export
  WAYLAND_DISPLAY=wayland-1
  ```

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

Run the application

The sample application uses the `/etc/configs/config_classification.json` file to read the input parameters.

To create your own config JSON file, use config_classification.json as a reference.

1. Use the following format of the `config_classification.json` file.

```
{
  "file-path": "<path-to-input-video>",
  "ml-framework": "<snpe or tflite or qnn framework>",
  "model": "<path-to-model-file>",
  "labels": "<path-to-label-file>",
  "threshold": <post processsing threshold, integer value from 1 to 100>,
  "constants": "<model constants for LiteRT Model>",
  "runtime": "<dsp, gpu, cpu runtime>"
}
```

   NOTE Update the config JSON file based on the model, input stream, and other properties. For more information, see Config JSON field description.

   For example, run the application using input from a file, LiteRT model, DSP runtime, custom constants, and custom threshold value:

```
{
"file-path": "/etc/media/video.mp4",
"ml-framework": "tflite",
"model": "/etc/models/inception_v3_quantized.tflite",
"labels": "/etc/labels/classification.labels",
"threshold": 40,
"runtime": "dsp",
"constants": "Inceptionv3,q-offsets=<38.0>,q-scales=<0.17039915919303894>;"
}
```

2. Run the gst-ai-classification application:

```
gst-ai-classification --config-file=/etc/configs/config_classification.json
```

To display the available help options, run the following command in the SSH shell:

```
gst-ai-classification -h
```

To stop the use case, use **CTRL + C**.

Expected output

The classified object is displayed on the local display.

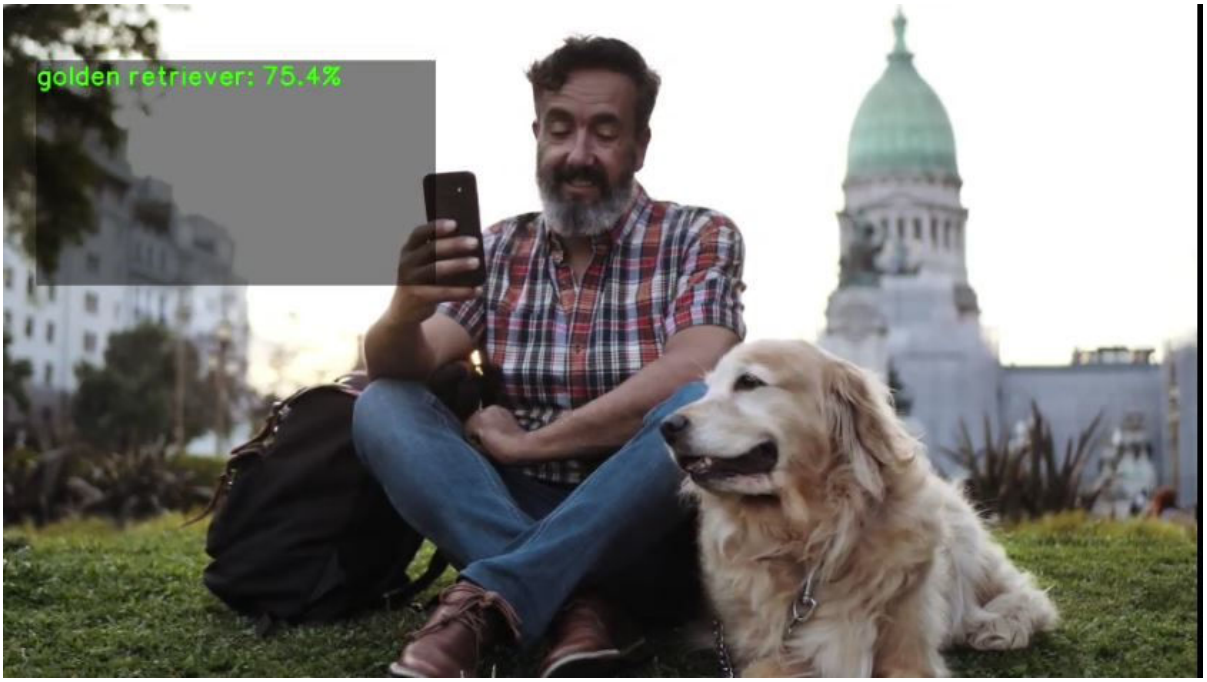


Figure 2-2 Expected output for gst-ai-classification application

Pipeline flow

The table lists the plugins used in the object classification pipeline:

Plugin	Description
Camera source:qtiqmmfsrc	▪ Captures the live stream from camera. ▪ Uses tee to split the stream for inferencing.
File source: filesrc	▪ Captures the video stream using filesrc, followed by qtdemux, which demultiplexes the stream. ▪ Uses tee to split the stream for inferencing.
RTSP source: rtspsrc	▪ Captures the RTSP stream using rtspsrc, followed by rtph264depay for video extraction. ▪ Uses tee to split the stream for inferencing.
h264parse	Parses the H.264 video.
v4l2h264dec	Decodes the video.
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs the following preprocessing on the stream data. This preprocessing is done when the model expects floating-point values as input. a. Color conversion b. Scaling (up or down) c. Normalization 3. Converts the preprocessed video stream to a tensor stream on its source pad. The tensor stream is used for inferencing in the later stages of the pipeline.

Plugin	Description
Inferencing plugins: ■ qtimlsnpe ■ qtimltflite ■ qtimlqnn	1. After the inference runtime receives the tensor stream on its sink pad, it runs the inference. 2. Produces a tensor stream with the inference results on its source pad.
qtimlvclassification	Handles inference results from any classification model. 1. Applies a threshold to the chosen number of results. For quantized model, add Softmax and constants (q-offsets and q-scales). 2. Loads the MobileNet postprocessing module. 3. Produces results as video frames with classification labels. 4. Sends these processed results to the sink pad of qtivcomposer.
qtivcomposer	1. Composes frames with contents from its sink pads. 2. Pushes the GStreamer buffers containing these composed frames to its source pad.
Waylandsink	1. Waylandsink submits the video stream received on its sink pad to Weston. 2. Weston renders the video stream on a local display.

Config JSON field description

Field	Values/description
ml-framework	Use one of the following models: ■ `snpe`: Qualcomm Neural Processing SDK ■ `tflite`: LiteRT ■ `qnn`: Qualcomm AI Engine direct
runtime	Use one of the following runtimes: ■ `cpu` ■ `gpu` ■ `dsp`
Input source	Use one of the following input sources: ■ `camera`: Primary (0) or secondary (1). ■ `file-path`: The directory path to the video file. ■ `rtsp-ip-port`: The address of the RTSP stream: *`rtsp://<ip>:<port>/<stream>`*

Known issues

The text overlay of the classification object is small.

RELATED INFORMATION

"Image classification and display with LiteRT" on page 213
"Image classification and encode with LiteRT" on page 217
"Image classification and display with Neural Processing SDK" on page 247
"Image classification and encode with Neural Processing SDK" on page 251

2.1.3 Object detection

The **gst-ai-object-detection** application allows you to detect objects within images and videos. The use cases show the execution of YOLOv5, YOLOv8, and YOLO-NAS using the Qualcomm Neural Processing SDK runtime, YOLOv8 using Qualcomm AI Engine direct, and YOLOv5 and YOLOv8 using LiteRT.

The figure shows the pipeline, which receives the input from a live camera feed, file, or an RTSP stream, performs preprocessing, runs inferences on AI hardware, and displays the results on the screen. For information about the plugins used in the pipeline flow, see Pipeline flow.

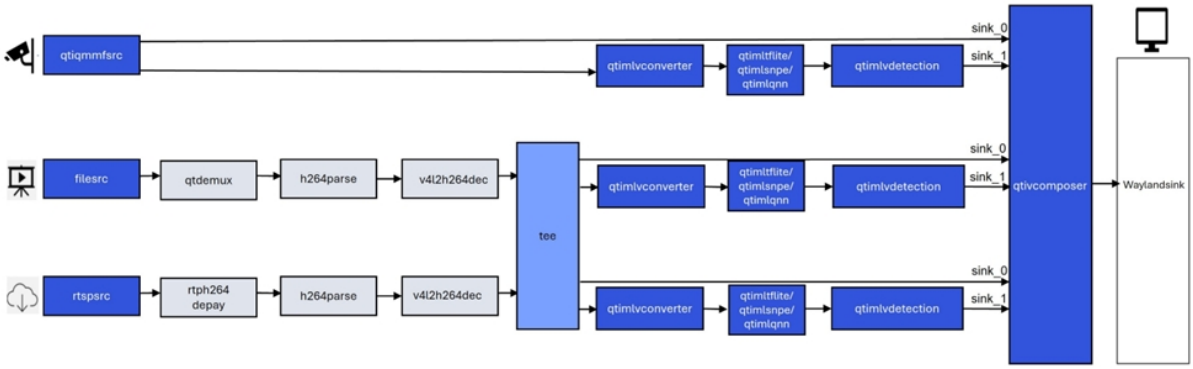


Figure 2-3 gst-ai-object-detection pipeline

This video guides you through setting up a live camera feed, implementing preprocessing steps, running AI inferences on dedicated hardware, and displaying the detection results live. The video will open in a new tab.

Sample model and label files

Table 2-2 Sample model and label files for gst-ai-object-detection

Runtime	Model files	Label files
Qualcomm Neural Processing SDK	*yolonas.dlc*	▪ *yolonas.labels* ▪ *yolov8.labels*
LiteRT	*yolov8_det_quantized.tflite*	
Qualcomm AI Engine direct	*yolov8_det_quantized.bin*	

NOTE The YOLOv8 Qualcomm AI Engine direct SDK model isn't supported in the current release.

Prerequisites

- If not already done so, Download and install eSDK.
- To run the application, push the model and label files to the device. For instructions, see Download model and label files.

  The application supports the Qualcomm Neural Processing SDK, Qualcomm AI Engine direct, and LiteRT models.
- To access your host computer, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is already enabled, you can skip this step.
- Push the model files from the Linux host computer:

  ```
  scp <model_filename> root@<IP address of target device>:/etc/models
  ```
- Note that the Download model and label files downloads the sample `video.mp4` video to the `/etc/media` directory. If you are using a custom video, then ensure that you push the video to `/etc/media` and update the file path in the application config.JSON file.
- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.
- Enable the display:

  ```
  export XDG_RUNTIME_DIR=/dev/socket/weston && export
  WAYLAND_DISPLAY=wayland-1
  ```

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

Run the application

The sample application uses the `/etc/configs/config_detection.json` file to read the input parameters.

To create your own config JSON file, use config_detection.json as a reference.

1. Enter the second SSH shell and copy the YOLO-NAS label files to YOLOv8:

   ```
   cp /etc/labels/yolonas.labels /etc/labels/yolov8.labels
   ```

2. Use the following format of the `/etc/configs/config_detection.json` file:

```
{
  "file-path": "<input video path>",
  "ml-framework": "<snpe, or tflite or qnn framework>",
  "yolo-model-type": "<yolov8 or yolonas or yolov5>",
  "model": "<Model Path>",
  "labels": "<Label Path>",
  "constants": "<Model Constants for LiteRT Model>",
  "threshold": <Post-processing threshold, integer value from 1-100>,
  "runtime": "<dsp, cpu or gpu runtime>"
}
```

NOTE Update the config JSON file based on the model, input stream, and other properties. For more information, see Config JSON field description.

For example, run the application using input from the video file, LiteRT, YOLOv8 model, DSP runtime, custom constants, and custom threshold value:

```
{
  "file-path": "/etc/media/video.mp4",
  "ml-framework": "tflite",
  "yolo-model-type": "yolov8",
  "model": "/etc/models/yolov8_det_quantized.tflite",
  "labels": "/etc/labels/yolov8.labels",
  "constants": "YOLOv8,q-offsets=<21.0, 0.0, 0.0>,q-
scales=<3.0546178817749023, 0.003793874057009816, 1.0>;",
  "threshold": 40,
  "runtime": "dsp"
}
```

3. Run the gst-ai-object-detection application:

```
gst-ai-object-detection --config-file=/etc/configs/config_detection.json
```

NOTE The `Yolo-NAS-Quantized.tflite` model can be run using the YOLOv8 value in the `yolo-model-type` field.

To display the available help options, run the following command in the SSH shell:

```
gst-ai-object-detection -h
```

To stop the use case, use **CTRL + C**.

Expected output

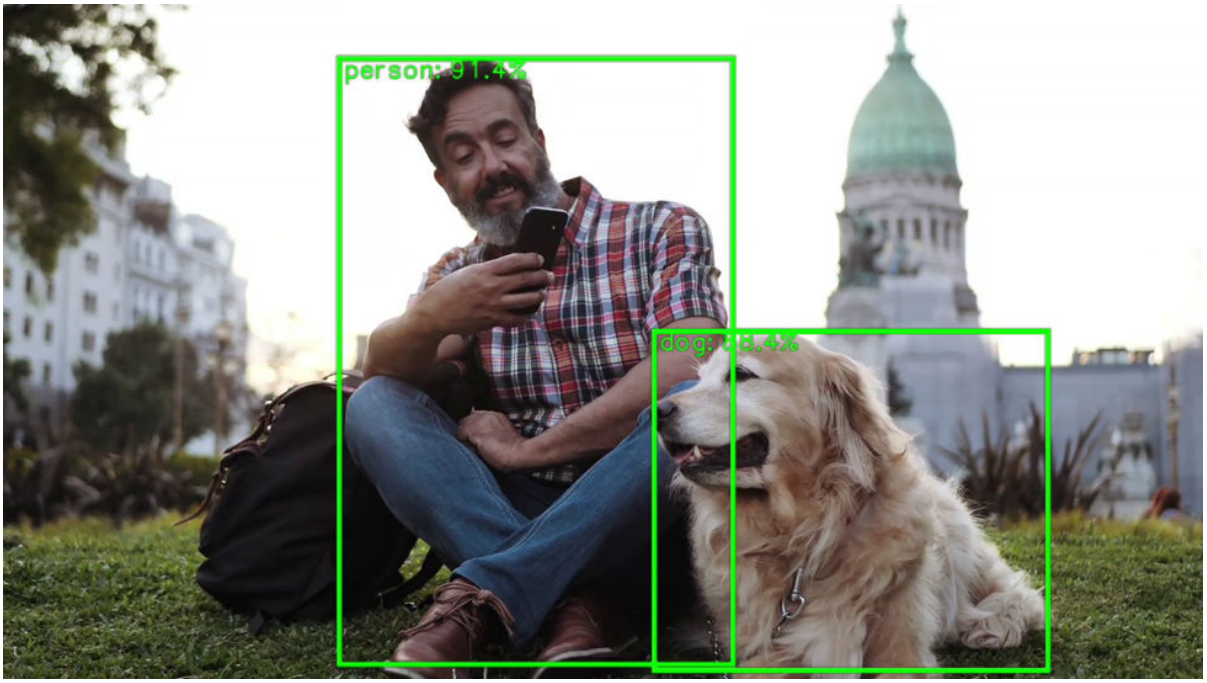


Figure 2-4 Expected output for gst-ai-object-detection application

Pipeline flow

The table lists the plugins used in the object detection pipeline:

Plugin	Description
Camera source:qtiqmmfsrc	▪ Captures the live stream from camera. ▪ Uses tee to split the stream for inferencing.
File source: filesrc	▪ Captures the video stream using filesrc, followed by qtdemux, which demultiplexes the stream. ▪ Uses tee to split the stream for inferencing.
RTSP source: rtspsrc	▪ Captures the RTSP stream using rtspsrc, followed by rtph264depay for video extraction. ▪ Uses tee to split the stream for inferencing.
h264parse	Parses the H.264 video.
v4l2h264dec	Decodes the video.
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs the following preprocessing on the stream data. This preprocessing is done when the model expects floating-point values as input. a. Color conversion b. Scaling (up or down) c. Normalization 3. Converts the preprocessed video stream to a tensor stream on its source pad. The tensor stream is used for inferencing in the later stages of the pipeline.

Plugin	Description
Inferencing plugins: ■ qtimlsnpe ■ qtimltflite ■ qtimlqnn	1. After the inference runtime receives the tensor stream on its sink pad, it runs the inference. 2. Produces a tensor stream with the inference results on its source pad.
qtimlvdetection	Handles inference results from any object detection model. 1. Applies a threshold to the chosen number of results. 2. Loads the YOLO (YOLOv5, YOLOv8, or YOLO-NAS) module. 3. Produces video frames with only bounding boxes that can be overlaid on objects. 4. Sends these processed frames to the sink pad of qtivcomposer.
qtivcomposer	1. Composes frames with contents from its sink pads. 2. Pushes the GStreamer buffers containing these composed frames to its source pad.
Waylandsink	1. Waylandsink submits the video stream received on its sink pad to Weston. 2. Weston renders the video stream on a local display.

Config JSON field description

Table 2-3 Field description–config_detection.json file

Field	Values/description
ml-framework	Use one of the following models: ■ `snpe`–Qualcomm Neural Processing SDK ■ `tflite`–LiteRT ■ `qnn`–Qualcomm AI Engine direct
yolo-model-type	Run the `yolov5`, `yolov8`, and `yolonas` models respectively. See Sample model and label files.
runtime	Use one of the following runtimes: ■ `cpu` ■ `gpu` ■ `dsp`
Input source	Use one of the following input sources: ■ `camera`–Primary (0) or secondary (1). ■ `file-path`–The directory path to the video file. ■ `rtsp-ip-port`–The address of the RTSP stream: *rtsp://<ip>:<port>/<stream>*

Known issues

The AI Hub YOLOv8 and YOLO-NAS AI Engine direct SDK models aren't supported. The models will be updated in a future release.

RELATED INFORMATION

"Object detection and display with LiteRT" on page 224
"Object detection and encode with LiteRT" on page 228
"Object detection and display with Neural Processing SDK" on page 255
"Object detection and encode with Neural Processing SDK" on page 259

2.1.4 Pose detection

The **gst-ai-pose-detection** application allows you to detect the body pose of the subject in an image or video. The use cases use an input stream from a camera, file, or an RTSP source, use LiteRT and Qualcomm AI Engine direct models for pose detection, and display the results on the screen.

The figure shows the pipeline, which receives the input from a live camera feed, file, or an RTSP stream, performs preprocessing, conducts inference on the AI hardware, and displays the results on the screen.

This process allows for real-time pose detection and visualization of human poses.

For information about the plugins used in the pipeline flow, see Pipeline flow.

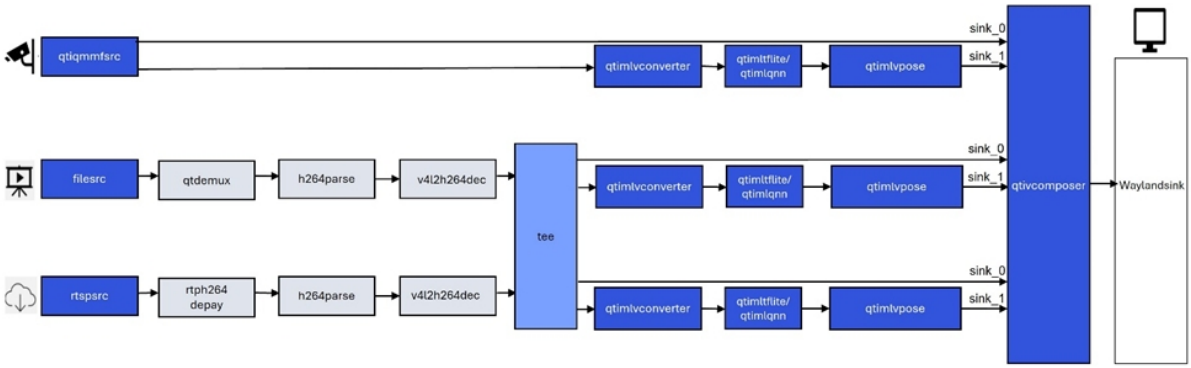


Figure 2-5 gst-ai-pose-detection pipeline

Sample model and label files

Table 2-4 Sample model and label files for gst-ai-pose-detection

Runtime	Model files	Label files
LiteRT	*hrnet_pose_quantized.tflite*	*hrnet_pose.labels*
Qualcomm AI Engine direct	*hrnet_pose_quantized.bin*	

Prerequisites

- If not already done so, Download and install eSDK.
- Download model and label files.
- To access your host computer, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is already enabled, you can skip this step.
- Push the model files from the Linux host computer.

  ```
  scp <model_filename> root@<IP address of target device>:/etc/models
  ```
- Note that the Download model and label files downloads the sample `video.mp4` video to the `/etc/media` directory. If you are using a custom video, then ensure that you push the video to `/etc/media` and update the file path in the application config.JSON file.

- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.
- Enable the display:

```
export XDG_RUNTIME_DIR=/dev/socket/weston && export
WAYLAND_DISPLAY=wayland-1
```

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

Run the application

The sample application uses the `/etc/configs/config_pose.json` file to read the input parameters.

To create your own config JSON file, use config_pose.json as a reference.

1. Use the following format of the `config_pose.json` file:

```
{
  "file-path": "<input-video-path>",
  "ml-framework": "<tflite or qnn framework>",
  "model": "<path-to-model-file",
  "labels": "<path-to-label-file",
  "constants": "<model-constants-for-LiteRT-model>",
  "threshold": <post-processing threshold, integer value from 1-100>,
  "runtime": ""<cpu, gpu or dsp runtime>"
}
```

NOTE Update the config JSON file based on the model, input stream, and other properties. For more information, see Config JSON field description.

For example, run the application using LiteRT, input video file, custom model, custom label file, DSP runtime, and custom threshold:

```
{
"file-path": "/etc/media/video.mp4",
"ml-framework": "tflite",
"model": "/etc/models/hrnet_pose_quantized.tflite",
"labels": "/etc/labels/hrnet_pose.labels",
"constants": "hrnet,q-offsets=<8.0>,q-scales=<0.0040499246679246426>;",
"threshold": 51,
"runtime": "dsp"
}
```

2. Run the gst-ai-pose-detection application:

```
gst-ai-pose-detection --config-file=/etc/configs/config_pose.json
```

To display the available help options, run the following command in the SSH shell:

```
gst-ai-pose-detection -h
```

To stop the use case, use **CTRL + C**.

Expected output

The displayed output shows the detected pose of the objects.

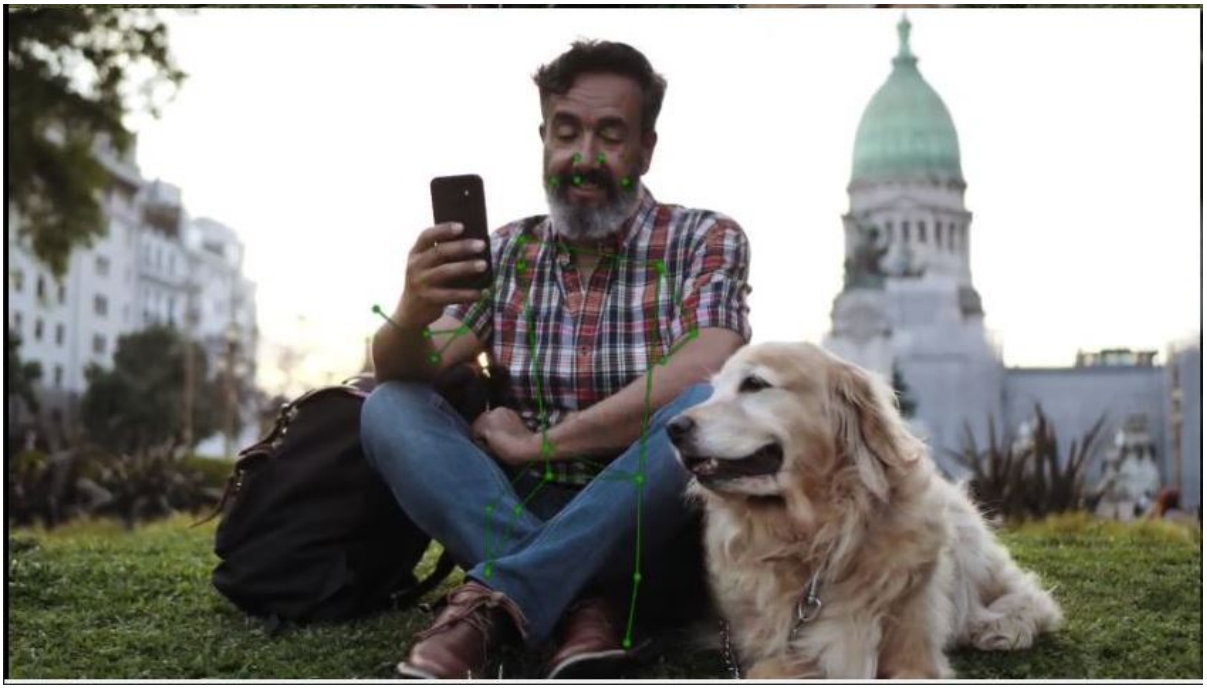


Figure 2-6 Expected output for gst-ai-pose-detection application

Pipeline flow

The table lists the plugins used in the pose detection pipeline:

Plugin	Description
Camera source:qtiqmmfsrc	▪ Captures the live stream from camera. ▪ Uses tee to split the stream for inferencing.
File source: filesrc	▪ Captures the video stream using filesrc, followed by qtdemux, which demultiplexes the stream. ▪ Uses tee to split the stream for inferencing.
RTSP source: rtspsrc	▪ Captures the RTSP stream using rtspsrc, followed by rtph264depay for video extraction. ▪ Uses tee to split the stream for inferencing.
h264parse	Parses the H.264 video.
v4l2h264dec	Decodes the video.
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs the following preprocessing on the stream data. This preprocessing is done when the model expects floating-point values as input. a. Color conversion b. Scaling (up or down) c. Normalization 3. Converts the preprocessed video stream to a tensor stream on its source pad.

Plugin	Description
	The tensor stream is used for inferencing in the later stages of the pipeline.
qtimltflite and qtimlqnn	Uses the HRNet model for pose detection. The application runs on the external delegate to run the model using the Hexagon Tensor Processor. After the inference runtime receives the tensor stream on its sink pad, qtimltflite does the following: ▪ Runs the inference. ▪ Produces a tensor stream containing the inference results, which is then made available on its source pad. ▪ Manages the inference results from any pose detection model.
qtimlvpose	▪ Applies a threshold to the chosen number of results. ▪ Loads corresponding modules for various pose detection models. In the use case described in this section, qtimlvpose does the following: a. Loads the HRNet module. b. Produces results in the form of video frames with drawn poses. c. Sends the results to the sink pad of qtivcomposer for further processing or display.
qtivcomposer	1. Composes frames by combining the contents from its sink pads. 2. Pushes the GStreamer buffers containing the composed frames onto its source pad.
Waylandsink	1. Forwards the video stream received on its sink pad to Weston. 2. Weston renders the video stream on a local display.

Config JSON field description

Table 2-5 Field description–config_pose.json file

Field	Values/description
ml-framework	Enable and use one of the following models: ▪ `tflite`: LiteRT ▪ `qnn`: Qualcomm AI Engine direct
runtime	Enable and use one of the following runtimes: ▪ `cpu` ▪ `gpu` ▪ `dsp`
Input source	Enable and use one of the following input sources: ▪ `camera`: Primary (0) or secondary (1). ▪ `file-path`: The directory path to the video file. ▪ `rtsp-ip-port`: The address of the RTSP stream: *rtsp://<ip>:<port>/<stream>*

Known issues

If the object is far from the camera, expect an accuracy drop.

RELATED INFORMATION

"Pose estimation and display with LiteRT" on page 236
"Pose estimation and encode with LiteRT" on page 240

2.1.5 Image segmentation

The **gst-ai-segmentation** application allows you to divide an image into different and meaningful parts or segments and assign a label to each homogenous segment based on the similarity of the attributes. Use Qualcomm Neural Processing SDK runtime, Qualcomm AI Engine direct runtime, and LiteRT for image segmentation.

The figure shows the pipeline, which receives the input from a live camera feed, file, or an RTSP stream, preprocesses the video data, runs inference using AI hardware, and displays the segmented data on the screen.

For information about the plugins used in the pipeline flow, see Pipeline flow.

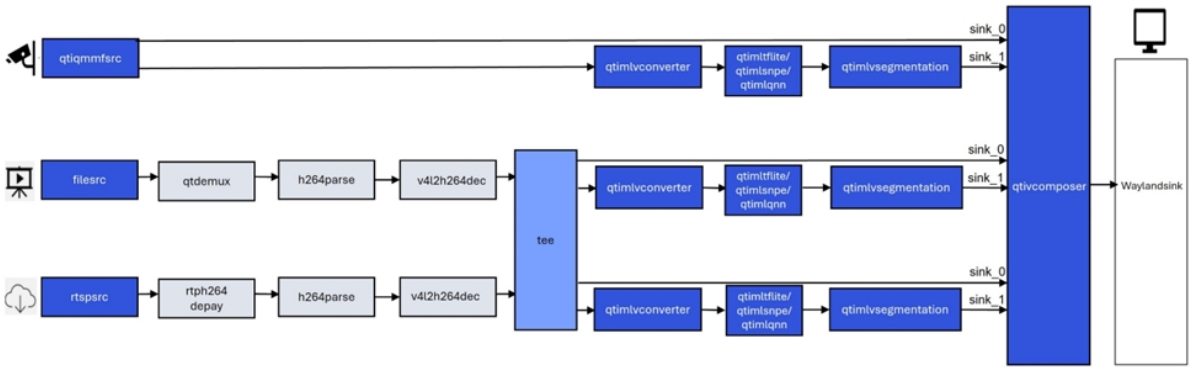


Figure 2-7 gst-ai-segmentation pipeline

Sample model and label files

Table 2-6 Sample model and label files for gst-ai-segmentation

Runtime	Model files	Label files
Qualcomm Neural Processing SDK	*deeplabv3_resnet50.dlc*	*deeplabv3_resnet50.labels*
LiteRT	*deeplabv3_plus_mobilenet_quantized.tflite*	
Qualcomm AI Engine direct	*deeplabv3_plus_mobilenet_quantized.bin*	

Prerequisites

- If not already done so, Download and install eSDK.
- To run the application, push the model and label files to the device. For instructions, see Download model and label files.

  The application supports the Qualcomm Neural Processing SDK, Qualcomm AI Engine direct, and LiteRT models.
- To access your host computer, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is already enabled, you can skip this step.
- Push the model files from the Linux host computer:

  ```
  scp <model_filename> root@<IP address of target device>:/etc/models
  ```
- Note that the Download model and label files downloads the sample `video.mp4` video to the `/etc/media` directory. If you are using a custom video, then ensure that you push the video to `/etc/media` and update the file path in the application config.JSON file.
- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.
- Enable the display:

  ```
  export XDG_RUNTIME_DIR=/dev/socket/weston && export
  WAYLAND_DISPLAY=wayland-1
  ```

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

Run the application

The sample application uses the `/etc/configs/config_segmentation.json` file to read the input parameters.

To create your own config JSON file, use config_segmentation.json as a reference.

1. Use the following format of the `config_segmentation.json` file:

   ```
   {
     "file-path": "<input-video-path>",
     "ml-framework": "<snpe, tflite, or qnn framework>",
     "model": "<path-to-model-file>",
     "labels": "<path-to-label-file>",
     "constants": "<model-constants-for-LiteRT-model>",
     "runtime": "<dsp, gpu, or cpu runtime>"
   }
   ```

   NOTE Update the config JSON file based on the model, input stream, and other properties. For more information, see Config JSON field description.

   For example, run the application using the `DeepLabV3-Plus-MobileNet` LiteRT model, DSP runtime, custom model and label paths, and constants from the video file:

   ```
   {
   "file-path": "/etc/media/video.mp4",
   "ml-framework": "tflite",
   ```

```
"model": "/etc/models/deeplabv3_plus_mobilenet_quantized.tflite",
"labels": "/etc/labels/deeplabv3_resnet50.labels",
"constants": "deeplab,q-offsets=<0.0>,q-scales=<1.0>;",
"runtime": "dsp"
}
```

NOTE The values for the `q-scales` and `q-offsets` constants are `<1.0>` and `<0.0>` respectively.

2. Run the gst-ai-segmentation application:

```
gst-ai-segmentation --config-file=/etc/configs/config_segmentation.json
```

To display the available help options, run the following command in the SSH shell:

```
gst-ai-segmentation -h
```

To stop the use case, use **CTRL + C**.

Expected output

The segmented data is displayed on the local display.

Figure 2-8 Expected output for gst-ai-segmentation application

Pipeline flow

The table lists the plugins used in the image segmentation pipeline:

Plugin	Description
Camera source:qtiqmmfsrc	■ Captures the live stream from camera. ■ Uses tee to split the stream for inferencing.
File source: filesrc	■ Captures the video stream using filesrc, followed by qtdemux, which demultiplexes the stream. ■ Uses tee to split the stream for inferencing.
RTSP source: rtspsrc	■ Captures the RTSP stream using rtspsrc, followed by rtph264depay for video extraction. ■ Uses tee to split the stream for inferencing.
h264parse	Parses the H.264 video.
v4l2h264dec	Decodes the video.
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs the following preprocessing on the stream data. This preprocessing is done when the model expects floating-point values as input. a. Color conversion b. Scaling (up or down) c. Normalization 3. Converts the preprocessed video stream to a tensor stream on its source pad. The tensor stream is used for inferencing in the later stages of the pipeline.
Inferencing plugins: ■ qtimlsnpe ■ qtimltflite ■ qtimlqnn	1. After the inference runtime receives the tensor stream on its sink pad, it runs the inference. 2. Produces a tensor stream with the inference results on its source pad.
qtimlvsegmentation	Converts the inference tensors that it receives on its sink pad into video formats that the multimedia plugins use for further processing.
qtivcomposer	1. Composes frames by combining content from its sink pads. 2. Pushes the GStreamer buffers containing these composed frames to its source pad.
Waylandsink	1. Forwards the video stream received on its sink pad to Weston. 2. Weston renders the video stream on a local display.

Config JSON field description

Table 2-7 Field description–config_segmentation.json file

Field	Values/description
ml-framework	Enable and use one of the following models: ■ `snpe`: Qualcomm Neural Processing SDK ■ `tflite`: LiteRT ■ `qnn`: Qualcomm AI Engine direct
runtime	Enable and use one of the following runtimes:

Table 2-7 Field description–config_segmentation.json file (cont.)

Field	Values/description
	▪ `cpu` ▪ `gpu` ▪ `dsp`
Input source	Enable and use one of the following input sources: ▪ `camera`: Primary (0) or secondary (1). ▪ `file-path`: The directory path to the video file. ▪ `rtsp-ip-port`: The address of the RTSP stream: *`rtsp://<ip>:<port>/<stream>`*

Known issues

An FPS drop is observed when using the Qualcomm Neural Processing Engine SDK models.

RELATED INFORMATION

"Image segmentation and display with LiteRT" on page 232
"Image segmentation and encode with LiteRT" on page 234
"Image segmentation and display with Neural Processing SDK" on page 263
"Image segmentation and encode with Neural Processing SDK" on page 265

2.1.6 Image segmentation using Python with container

The application allows you to perform image segmentation using the Qualcomm Neural Processing SDK with Python bindings, all from within a docker container.

Prerequisites

Ensure that the following are available:

- Ensure that you are using a Linux host computer with a docker installed.
- Download the required model files on the device. For instructions, see Download model and label files.

Figure 2-9 Workflow to set up host container

To set up the host container, do the following:

1. Install Qualcomm® Package Manager (QPM3).
2. Download and extract Qualcomm Neural Processing SDK:

```
qpm-cli --login <username>
```

3. Download scripts and model attachments to run the sample mode.
   a. Download the Dockerfile and scripts and prepare the directory for storing the image.

      ```
      git clone https://git.codelinaro.org/clo/le/sdk-tools.git -b imsdk-tools.lnx.1.0
      ```

      ```
      cd sdk-tools/snpe-container-python
      ```

      ```
      mkdir images
      ```

      The `snpe-container-python` file has the Dockerfile and scripts. Run all the commands from this directory.

   b. Copy the test image in a new folder called `inputs` and rename the test image as `input_image.jpg`.

      ```
      mkdir inputs
      ```

      ```
      cp <path of test image> ./inputs
      ```

      ```
      cd inputs
      ```

      ```
      mv <test_image_name> input_image.jpg
      ```

   c. Set up the host device for cross compilation:

      ```
      sudo groupadd docker
      ```

      ```
      sudo usermod -aG docker $USER
      ```

      ```
      newgrp docker
      ```

      Until the host device is rebooted, continue running these commands for every new console that's used to run the docker.

      ```
      sudo apt-get install qemu-user-static qemu-system-arm
      ```

      ```
      docker run --rm --privileged multiarch/qemu-user-static --reset -p yes
      ```

      ```
      docker buildx rm builder
      ```

      ```
      docker buildx create --name builder --driver docker-container --use
      ```

      ```
      docker buildx inspect --bootstrap
      ```

   d. Build the docker image. Populate the `<path-to-sdk-tools>/targets/config.json` file.

      The following code shows an updated sample `config.json` file.

      The following inputs are mandatory:

      - `SNPE_version`
      - `Base_Image`

- `Target_platform`: The following are field values for each SoC:
  - For QCS6490–`qcm6490`
  - For QCS8275–`qcs8300`
  - For QCS9075–`qcs9100`
- `URL` (ensure that the address provided isn't the same as your current directory)

NOTE To prevent code failures, ensure that you remove the comments before using it.

```
{
    "SNPE_version":"v2.32.0.250228",
    "Base_Image": "ubuntu:22.04",
    "Target_platform": "qcm6490",

    "Additional_tag_container": "",
    "Additional_tag_image": "",
    "URL": "<path-to-sdk-tools>/snpe-container-python/images",

    "DeviceID" : null
}
```

4. Build the docker image:

   ```
   sudo apt install jq
   cd  <path to sdk-tools>/snpe-container-python
   source scripts/host/docker_env_setup.sh
   qml-docker-build-image targets/config.json
   ```

   Save the docker image:

   ```
   qml-docker-device-save-image targets/config.json
   ```

   The docker image is compressed and the TAR file is saved at the directory specified in the `URL` field in `config.json`.

   - If `Additional_tag_image` is empty, the compressed image is stored as `qml.tar`.
   - If `Additional_tag_image` is populated, the compressed image is stored as `qml-<field value>.tar`.

5. Push the docker image on the device:

   ```
   scp <path specified in URL field>/qml.tar root@<IP address of target device>:/opt
   ```

6. Start the container:

   ```
   ssh root@<ip of target>
   mount -o remount, rw /usr
   systemctl restart docker
   docker load < /opt/qml.tar
   ```

7. Run the docker image on the device:

```
docker run -it -d --device=/dev/fastrpc-cdsp-secure \
--device /dev/kgsl-3d0 --device /dev/dma_heap/system --device /dev/
dma_heap/qcom,system \
-v /usr/lib/libCB.so:/usr/lib/libCB.so \
-v /usr/lib/libOpenCL.so:/usr/lib/libOpenCL.so \
-v /usr/lib/libOpenCL_adreno.so:/usr/lib/libOpenCL_adreno.so \
-v /usr/lib/libbase.so.0:/usr/lib/libbase.so.0 \
-v /usr/lib/libcdsprpc.so:/usr/lib/libcdsprpc.so \
-v /usr/lib/libcutils.so.0:/usr/lib/libcutils.so.0 \
-v /usr/lib/libdmabufheap.so.0:/usr/lib/libdmabufheap.so.0 \
-v /usr/lib/libglib-2.0.so.0:/usr/lib/libglib-2.0.so.0 \
-v /usr/lib/libgsl.so:/usr/lib/libgsl.so \
-v /usr/lib/libgthread-2.0.so.0:/usr/lib/libgthread-2.0.so.0 \
-v /usr/lib/libion.so.0:/usr/lib/libion.so.0 \
-v /usr/lib/libllvm-qcom.so:/usr/lib/libllvm-qcom.so \
-v /usr/lib/liblog.so.0:/usr/lib/liblog.so.0 \
-v /usr/lib/libpcre.so.1:/usr/lib/libpcre.so.1 \
-v /usr/lib/libsync.so.0:/usr/lib/libsync.so.0 \
-v /usr/lib/libvmmem.so.0:/usr/lib/libvmmem.so.0 \
-v /usr/lib/libatomic.so.1:/usr/lib/libatomic.so.1 \
-h qml --name qml qml
```

A docker image called `qml` should be running. To check this image, run the following command:

```
docker ps
```

8. Push `input_image.jpg` to the device.

```
scp <path to inputs_directory>/input_image.jpg root@<IP address of target
device>:/opt/
```

```
ssh root@<ip of target>
```

```
docker cp /opt/input_image.jpg qml:/opt/
```

9. Copy the `deeplab_resnet50.dlc` model to the docker:

```
docker cp /etc/models/deeplabv3_resnet50.dlc qml:/opt/
```

Run the application

1. Run the Qualcomm Neural Processing SDK model using Python bindings:

```
docker exec qml python3 /mnt/qml/src/python/snpe/test_snpe/
snpe_segmentation_app.py -d /opt/deeplabv3_resnet50.dlc -i /opt/
input_image.jpg -r dsp -o /mnt/qml/output/ -b USERBUFFER_FLOAT -p /usr/lib/
libSNPE.so
```

   - The output image is saved in the container at `/opt/`.
   - The output from the DLC model (RAW file) is saved at `/mnt/qml/output/`.

2. Copy the output from the container to the device:

```
docker cp qml:/opt/output.jpg /opt/
```

3. To pull the image from the device to host, run the command on your host computer:

```
scp root@<IP address of target device>:/opt/output.jpg ./
```

Expected output

Figure 2-10 Expected output for image segmentation using Python binding

2.1.7 Parallel inferencing

The **gst-ai-parallel-inference** application allows you to perform object detection, object classification, pose detection, and image segmentation on an input stream from different sources such as a camera, a file, or an RTSP network. The use cases use the LiteRT models for object detection, image segmentation, classification, and pose detection.

NOTE This application isn't supported on QCS8275.

The figure shows the pipeline, which takes the input from a camera, file, or an RTSP stream, performs the parallel inferencing for the four use cases, and display the results side by side on the screen.

For information about the plugins used in the pipeline flow, see Pipeline flow.

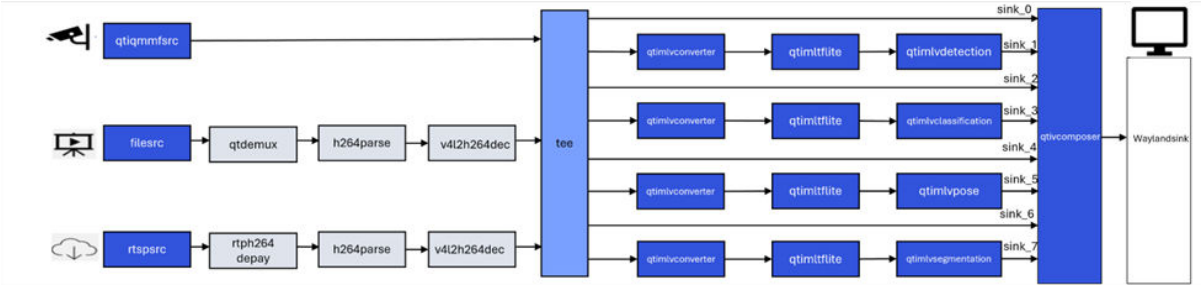


Figure 2-11 gst-ai-parallel-inference pipeline

Enhance your development projects by performing the four parallel AI inferences on a live camera stream. This video shows the step-by-step procedure on how to set up the pipeline and display the results side by side. The video will open in a new tab.

Sample model and label files

Table 2-8 Sample model and label files for gst-ai-parallel-inference

Application	Model files
Object detection	*yolov8_det_quantized.tflite*
Pose estimation	*hrnet_pose_quantized.tflite*
Segmentation	*deeplabv3_plus_mobilenet_quantized.tflite*
Classification	*inception_v3_quantized.tflite*

Prerequisites

- If not already done so, Download and install eSDK.
- Download model and label files.
- To access your host computer, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is already enabled, you can skip this step.
- Push the model files from the Linux host computer.

  ```
  scp <model_filename> root@<IP address of target device>:/etc/models
  ```
- Note that the Download model and label files downloads the sample `video.mp4` video to the `/etc/media` directory. If you are using a custom video, then ensure that you push the video to `/etc/media` and update the file path in the application config.JSON file.
- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.
- Enable the display:

  ```
  export XDG_RUNTIME_DIR=/dev/socket/weston && export
  WAYLAND_DISPLAY=wayland-1
  ```

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

Run the application

For sample model and label files, see Sample model and label files.

Enter SSH shell and copy the YOLO-NAS label files to YOLOv8:

```
cp /etc/labels/yolonas.labels /etc/labels/yolov8.labels
```

To run the application, use the following syntax:

```
gst-ai-parallel-inference [OPTION?]
```

- Run with input from the primary and secondary cameras:

  ```
  gst-ai-parallel-inference -c 0
  ```

  ```
  gst-ai-parallel-inference -c 1
  ```
- Run with input from a file:

  ```
  gst-ai-parallel-inference -s /etc/media/video.mp4
  ```
- Run with input from an RTSP stream:

  ```
  gst-ai-parallel-inference --rtsp-ip-port="rtsp://<ip-address-of-device>/
  live.mkv"
  ```
- Run with input from a video stream with constants for the LiteRT models for all four inferences:

  ```
  gst-ai-parallel-inference --file-path="/etc/media/video.mp4" --object-
  detection-constants="YOLOv8,q-offsets=<21.0, 0.0, 0.0>,q-
  scales=<3.0546178817749023, 0.003793874057009816, 1.0>;" --pose-detection-
  constants="Posenet,q-offsets=<8.0>,q-scales=<0.0040499246679246426>;" --
  segmentation-constants="deeplab,q-offsets=<0.0>,q-scales=<1.0>;" --
  classification-constants="Inceptionv3,q-offsets=<38.0>,q-
  scales=<0.17039915919303894>;"
  ```

To display the available help options, run the following command in the SSH shell:

```
gst-ai-parallel-inference -h
```

To stop the use case, use **CTRL + C**.

Expected output

After performing the four parallel inferences, the results are displayed side by side on the screen.

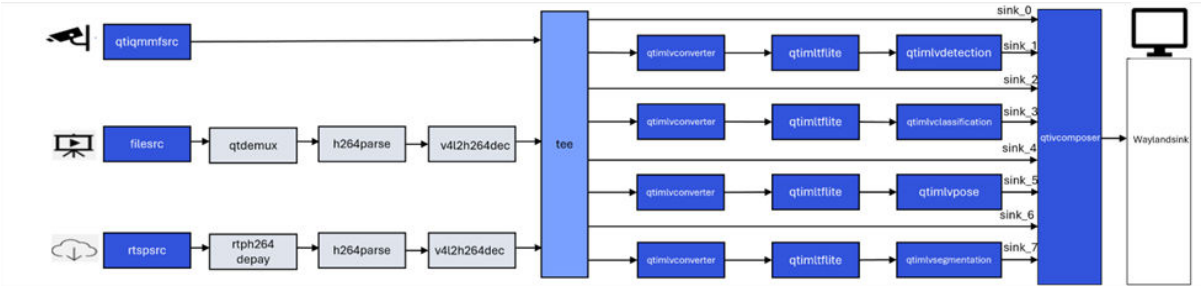


Figure 2-12 Expected output for gst-ai-parallel-inference application

Pipeline flow

The table lists the plugins used in the parallel inference pipeline:

Plugin	Description
Camera source:qtiqmmfsrc	▪ Captures the live stream from camera. ▪ Uses tee to split the stream for inferencing.
File source: filesrc	▪ Captures the video stream using filesrc, followed by qtdemux, which demultiplexes the stream. ▪ Uses tee to split the stream for inferencing.
RTSP source: rtspsrc	▪ Captures the RTSP stream using rtspsrc, followed by rtph264depay for video extraction. ▪ Uses tee to split the stream for inferencing.
h264parse	Parses the H.264 video.
v4l2h264dec	Decodes the video
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs the following preprocessing on the stream data. This preprocessing is done when the model expects floating-point values as input. a. Color conversion b. Scaling (up or down) c. Normalization The tensor stream is used for inferencing in the later stages of the pipeline.
qtimltflite	1. After the inference runtime receives the tensor stream on its sink pad, it runs the inference. 2. Produces a tensor stream with the inference results on its source pad.
Postprocessing plugins	Handles the inference results from any object detection, classification, pose detection, and segmentation model.

Plugin	Description
	■ qtimlvdetection a. Applies a threshold to the chosen number of results. b. Loads the YOLO-NAS module. c. Produces video frames with only bounding boxes that can be overlaid on objects, sending them to the sink pad of the qtivcomposer. ■ qtimlvclassification a. Applies the threshold to the chosen number of results. b. Loads the MobileNet module. c. Produces results as video frames with classification labels, sending them to the sink pad of the qtivcomposer. ■ qtimlvpose a. Applies the threshold to the chosen number of results. b. Loads the corresponding modules for various pose estimation models. For the use cases described in this section, qtimlvpose loads the PoseNet module. c. Produces results as video frames with poses drawn, sending them to the sink pad of the qtivcomposer. ■ qtimlvsegmentation: Converts the inference tensors that it receives on its sink pad into video formats that the multimedia plugins for further processing.
qtivcomposer	1. Composes frames with contents from its sink pads. 2. Pushes the GStreamer buffers containing these composed frames to its source pad.
Waylandsink	1. Waylandsink submits the video stream received on its sink pad to Weston. 2. Weston renders the video stream on a local display.

RELATED INFORMATION

"Parallel inference using Python" on page 199
"Image classification" on page 28
"Object detection" on page 33
"Pose detection" on page 38
"Image segmentation" on page 42

2.1.8 Multi input/output object detection

The **gst-ai-multi-input-output-object-detection** application allows you to perform objection detection on video streams from various sources such as a camera, a file, or over a network such as RTSP.

The figure shows the pipeline workflow, which captures video streams for inferencing from different sources such as camera, file, or RTSP. For information about the plugins used in the pipeline, see Pipeline flow.

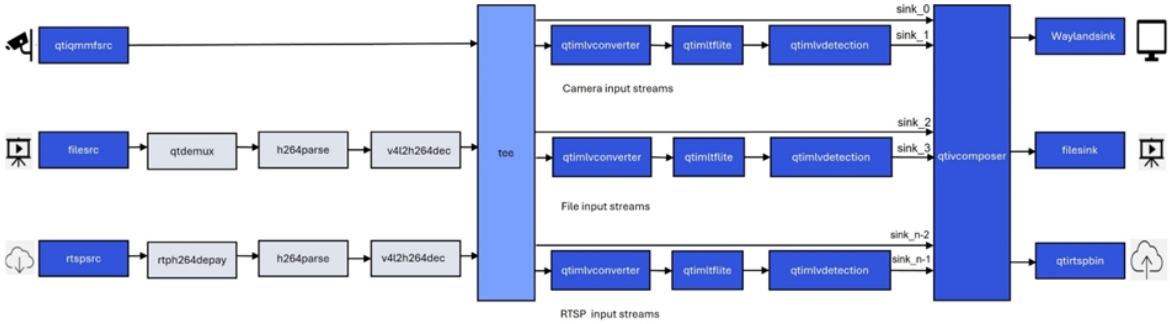


Figure 2-13 Multi-input inferencing pipeline

Sample model and label files

Table 2-9 Sample model and label files for gst-ai-multi-input-output-object-detection

Runtime	Model files	Label files
LiteRT	*yolov5.tflite*	*yolov5.labels*

Prerequisites

1. If not already done so, Download and install eSDK.
2. Download the YOLOv5 model.

   Running the `yolov5.tflite` model requires a Python environment.

   NOTE Update the following commands according to the Python version in your host environment.

3. Create the Python 3.8 virtual environment:

```
sudo apt-get install python3.8
python3.8 -m venv py3.8
source py3.8/bin/activate
```

4. Generate the `yolov5.tflite` model:

```
git clone https://github.com/ultralytics/yolov5.git
cd yolov5
python -m pip install -r requirements.txt tensorflow-cpu
python export.py --weights yolov5m.pt --img 320 --include tflite --int8 --
data data/coco128.yaml
```

5. On the host computer, push the model to the device:

   ```
   scp yolov5m-int8.tflite root@<IP address of the device>:/etc/models/
   yolov5.tflite
   ```

6. Download model and label files.
7. Enable SSH to access your host device. For instructions, see Sign in using SSH.

   NOTE If SSH is already enabled, you can skip this step.

8. Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.
9. Enable the display:

   ```
   export XDG_RUNTIME_DIR=/dev/socket/weston && export
   WAYLAND_DISPLAY=wayland-1
   ```

10. Push the model files from the Linux host computer:

    ```
    scp <model_filename> root@<IP address of target device>:/etc/models
    ```

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

Run the application

YOLO-NAS and YOLOv5 models are trained on the same Common Objects in Context (COCO) dataset.

Enter SSH shell and copy the YOLO-NAS label files to YOLOv5:

```
cp /etc/labels/yolonas.labels /etc/labels/yolov5.labels
```

NOTE The following commands provide the default model and label paths. If you have a different folder structure, replace the default paths in the command-line parameters. See Sample model and label files.

- Run object detection on streams from two cameras:

  ```
  gst-ai-multi-input-output-object-detection --num-camera=2 --display --
  model=/etc/models/yolov5.tflite --labels=/etc/labels/yolov5.labels
  ```

- Run object detection on streams from a file source and display the output:

  ```
  gst-ai-multi-input-output-object-detection --num-file=2 --display --
  model=/etc/models/yolov5.tflite --labels=/etc/labels/yolov5.labels
  ```

  The default source paths for the video files are `/etc/media/video1.mp4` and `/etc/media/video2.mp4`.

- Run object detection on streams from an RTSP source and display the output:

  ```
  gst-ai-multi-input-output-object-detection --num-rtsp=2 --rtsp-ip-
  port=<ip:port> --display --model=/etc/models/yolov5.tflite --labels=/etc/
  labels/yolov5.labels
  ```

  It's expected to have the `live1.mkv` and `live2.mkv` streams on for `ip-port`.

- Take the input source video stream from a file and display the output as follows:
  - Display the output on a screen.
  - Save the output on a file.
  - Send the output over the network through RTSP streaming.

  Before running this use case, ensure that the evaluation kit (EVK) and the host computer are connected to the same network:

  Run the use cases in a shell:

  ```
  gst-ai-multi-input-output-object-detection --num-file=2 -d -f /etc/media/
  app.mp4 --out-rtsp --model=/etc/models/yolov5.tflite --labels=/etc/labels/
  yolov5.labels -i <evk-ip-address> -p <port_num>
  ```

  To view the RTSP stream on the host PC, do the following:

  - Install VLC Media Player on the host computer and set the environment variables.
  - On the Linux host computer, do one of the following to see the RTSP stream:

    ```
    vlc -vvv rtsp://<evk-ip-address>:<port_num>/live
    ```

    ```
    ffplay -rtsp_transport tcp rtsp://<evk-ip-address>:<port_num>/live
    ```

  - On the Windows host computer, do the following:
    i. Open the VLC Media Player.
    ii. Select **Media** > **Open Network Stream** or use **CTRL + N**).
    iii. Enter `rtsp://<evk-ip-address>:<port_num>/live`.
    iv. Select **Play**.

    NOTE Ensure that the total number of input streams from the camera, RTSP, and file source doesn't exceed 6.

To display the available help options, run the following command in the SSH shell:

```
gst-ai-multi-input-output-object-detection --help
```

To stop the use case, use **CTRL + C**.

Expected output

Based on the use case, the results are either displayed on an HDMI screen, saved as an H.264 encoded MP4 file, or streamed over the RTSP server.

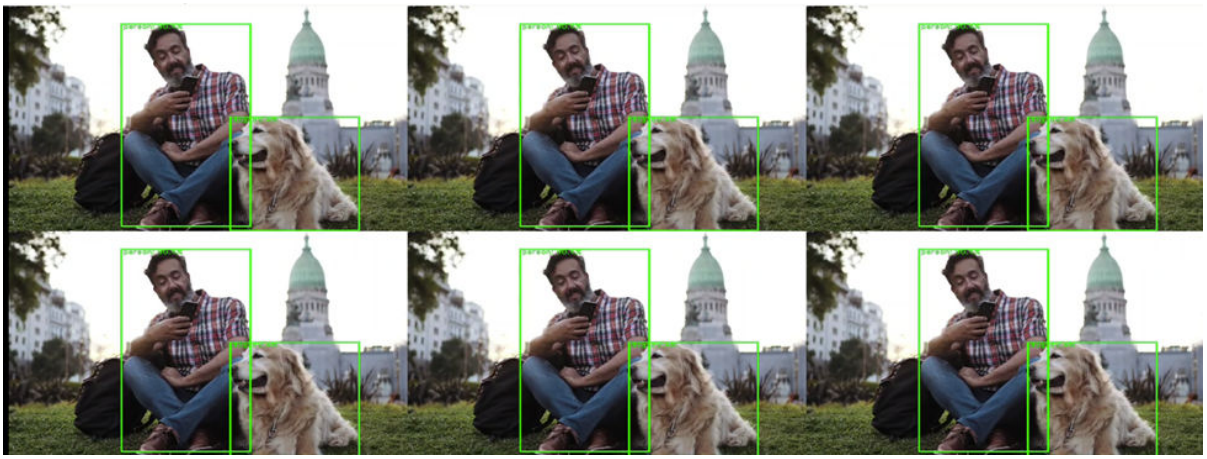


Figure 2-14 Expected output for gst-ai-multi-input-output-object-detection application–Preview

Pipeline flow

The table lists the plugins used in the multi input/output inference use cases:

Plugin	Description
Camera source:qtiqmmfsrc	▪ Captures the live stream from camera. ▪ Uses tee to split the stream for inferencing.
File source: filesrc	▪ Captures the video stream using filesrc, followed by qtdemux, which demultiplexes the stream. ▪ Uses tee to split the stream for inferencing.
RTSP source: rtspsrc	▪ Captures the RTSP stream using rtspsrc, followed by rtph264depay for video extraction. ▪ Uses tee to split the stream for inferencing.
h264parse	Parses the H.264 video.
v4l2h264dec	Decodes the video.
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs the following preprocessing on the stream data. This is done when the model expects floating-point values as input. a. Color conversion b. Scaling (up or down) c. Normalization The tensor stream is used for inferencing in the later stages of the pipeline.
qtimltflite	Runs on the LiteRT and uses the `yolov5.tflite` model for object detection. 1. After the inference runtime receives the tensor stream on its sink pad, it runs the inference. 2. Produces a tensor stream with the inference results on its source pad.

Plugin	Description
qtimlvdetection	Converts the inference tensors that it receives on its sink pad into video formats that the multimedia plugins can process later.
qtivcomposer	1. Composes frames with contents from its sink pads. 2. Pushes the GStreamer buffers containing these composed frames to its source pad.
Waylandsink	1. Waylandsink submits the video stream received on its sink pad to Wayland compositor. 2. Renders the video stream on a local display.
Filesink	Takes the video stream that it receives on its sink pad and saves it as an H.264-encoded MP4 file.
qtirtspbin	1. Serves as a network sink. 2. Transmits UDP packets to the network.

Pull the files from the target device:

```
scp root@<IP address of target device>:/etc/media/out.mp4 <destination
directory>
```

Known issues

- An fps drop is observed when running the application with six input streams.
- A display crash is observed when running two cameras together in a long run scenario.

RELATED INFORMATION

"Object detection" on page 33

2.1.9 Daisy chain detection and classification

The **gst-ai-daisychain-detection-classification** application allows you to perform cascaded object detection and classification with a camera, file source, or RTSP stream. The use case involves detecting objects and classifying the detected objects.

The figures show the pipeline workflow, which captures the video stream from the source, preprocesses the video, runs inference using AI hardware, and displays the results on the screen.

For information about the plugins used in this pipeline, see Pipeline flow.

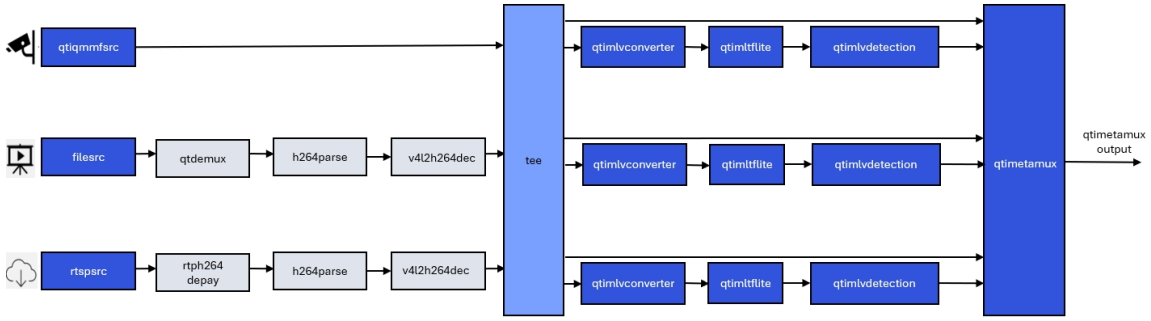


Figure 2-15 gst-ai-daisychain-detection-classification source pipeline

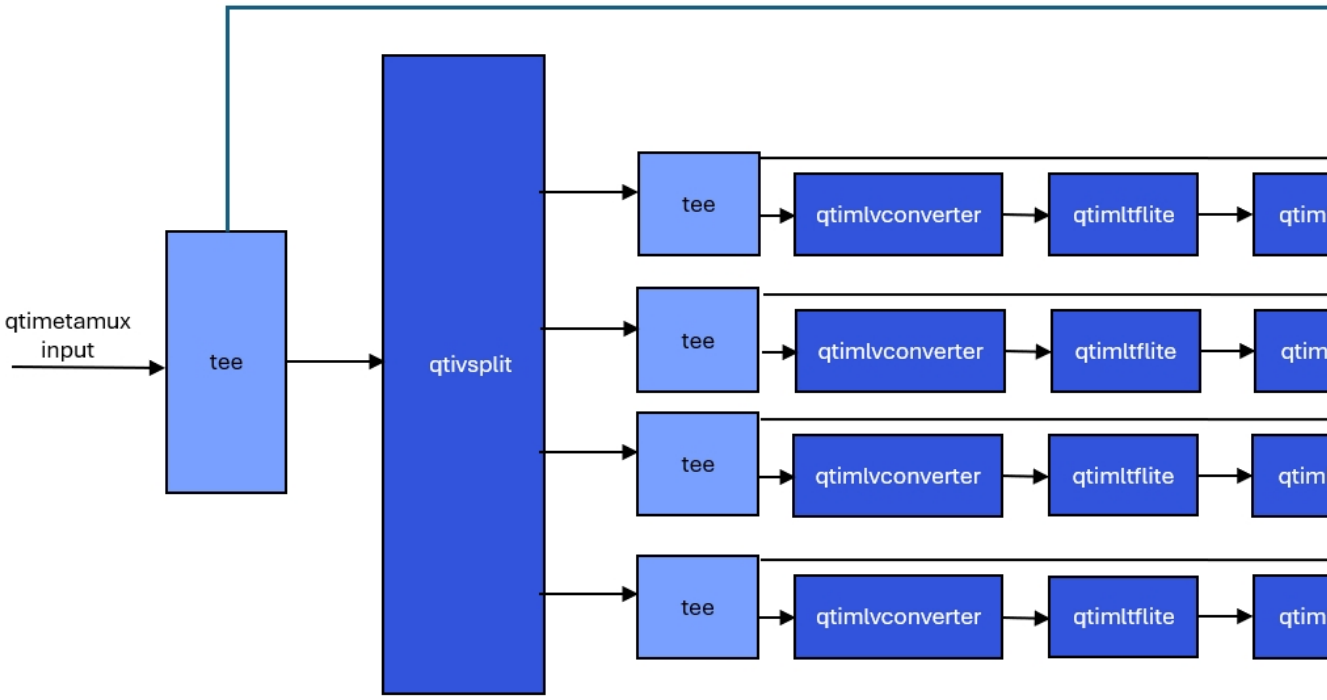


Figure 2-16 gst-ai-daisychain-detection-classification inference pipeline

Sample model and label files

Table 2-10 Sample model and label files for gst-ai-daisychain-detection-classification

Runtime	Model files	Label files
LiteRT	▪ Detection: *yolov8_det_quantized.tflite* ▪ Classification: *mobilenet_v2_quantized.tflite*	▪ Detection: *coco_labels.txt* ▪ Classification: *imagenet_labels.txt*

Prerequisites

- If not already done so, Download and install eSDK.
- Download model and label files.
- To access your host computer, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is already enabled, you can skip this step.
- Push the model files from the Linux host computer.

  ```
  scp <model_filename> root@<IP address of target device>:/etc/models
  ```
- Note that the Download model and label files downloads the sample `video.mp4` video to the `/etc/media` directory. If you are using a custom video, then ensure that you push the video to `/etc/media` and update the file path in the application config.JSON file.

- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.
- Enable the display:

```
export XDG_RUNTIME_DIR=/dev/socket/weston && export
WAYLAND_DISPLAY=wayland-1
```

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

Run the application

The sample application uses the `/etc/configs/config_daisychain_detection_classification.json` file to read the input parameters.

To create your own config JSON file, use config_daisychain_detection_classification.json as a reference.

1. Use the following format of the `config_daisychain_detection_classification.json` file:

```
{
  "input-file": "<input-video-path>",
  "detection-model": "<path-to-detection model>",
  "detection-labels": "<path-to-detection-labels>",
  "classification-model": "<path-to-classification-model>",
  "classification-labels": "<path-to-classification-labels>",
  "detection-constants": "<detection-LiteRT-model-constants>",
  "classification-constants": "<classification-litert-model-constants>"
}
```

NOTE
- Update the config JSON file based on the model, input stream, and other properties. For more information, see Config JSON field description.
- For QCS6490, if `file-path` and `rtsp-ip-port` are *not* present in the configuration file, then the camera input is selected.

For example, run the application using the custom video input file, model and label paths, and constants:

```
{
  "input-file": "/etc/media/video.mp4",
  "detection-model": "/etc/models/yolov8_det_quantized.tflite",
  "detection-labels": "/etc/labels/coco_labels.txt",
  "classification-model": "/etc/models/mobilenet_v2_quantized.tflite",
  "classification-labels": "/etc/labels/imagenet_labels.txt",
  "detection-constants": "YOLOv8,q-offsets=<21.0, 0.0, 0.0>,q-
scales=<3.0546178817749023, 0.003793874057009816, 1.0>;",
"classification-constants": "Inceptionv3,q-offsets=<38.0>,q-
scales=<0.17039915919303894>;"
}
```

2. Run the gst-ai-daisychain-detection-classification application:

```
gst-ai-daisychain-detection-classification --config-file=/etc/configs/
config_daisychain_detection_classification.json
```

To display the available help options, run the following command in the SSH shell:

```
gst-ai-daisychain-detection-classification -h
```

To stop the use case, use **CTRL + C**.

Expected output

The cropped video frame is overlaid on the frame and displayed on a local device.

NOTE The classification models trained on the Imagenet dataset don't contain the *person* class.

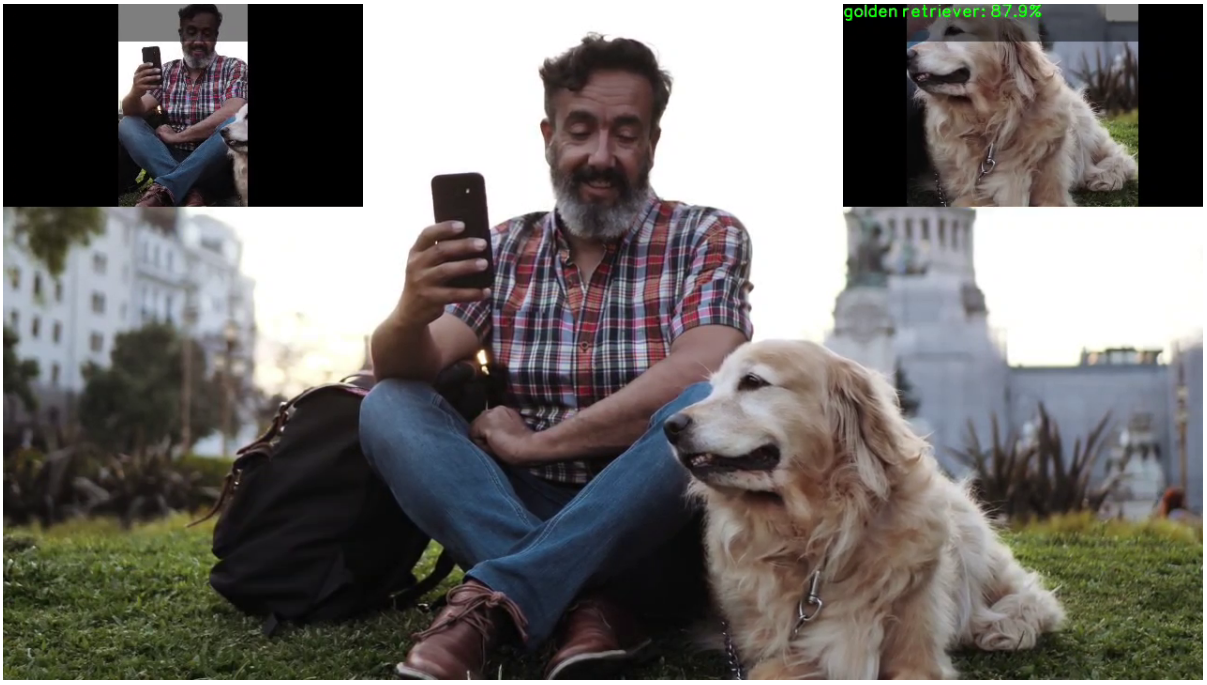


Figure 2-17 Expected output for gst-ai-daisychain-detection-classification application

Pipeline flow

The table lists the plugins used in the daisy chain detection and classification pipeline:

Plugin	Description
Camera source:qtiqmmfsrc	▪ Captures the live stream from camera. ▪ Uses tee to split the stream for inferencing.
File source: filesrc	▪ Captures the video stream using filesrc, followed by qtdemux, which demultiplexes the stream. ▪ Uses tee to split the stream for inferencing.
RTSP source: rtspsrc	▪ Captures the RTSP stream using rtspsrc, followed by rtph264depay for video extraction. ▪ Uses tee to split the stream for inferencing.
h264parse	Parses the H.264 video.
v4l2h264dec	Decodes the video.
qtimetamux	Multiplexes the stream.
h264parse	Parses the video.
v4l2h264dec	Decodes the H.264 video stream. The stream is forwarded to the Wayland compositor and is rendered on a local display device.
qtivsplit	Crops full frame into smaller frames based on the detected bounding boxes detected (maximum 4).
qtimlvconverter	Used by AI processing stream for preprocessing:

Plugin	Description
	1. Receives the video stream on its sink pad. 2. Performs the following preprocessing on the stream data. This preprocessing is done when the model expects the floating-point values as an input. a. Color conversion b. Scaling (up or down) c. Normalization 3. Converts the preprocessed video stream to a tensor stream on its source pad. The tensor stream is used for inferencing in the later stages of the pipeline.
qtimltflite	1. After the inference runtime receives the tensor stream on its sink pad, it runs the inference. 2. Produces a tensor stream with the inference results on its source pad.
qtimlvdetection	Handles inference results from any classification model. 1. Applies a threshold to the chosen number of results. 2. Loads the YOLOv5 module. 3. Generates results in the form of video frames with classification labels. 4. Produces video frames with only bounding boxes that can be cropped.
qtimlvclassification	1. Applies the threshold to the chosen number of results on the cropped frame. 2. Loads MobileNet module. 3. Produces results as video frames with classification labels. 4. Sends them to the sink pad of qtivcomposer.
Waylandsink	Submits the video stream that it's receiving on its sink pad to the Wayland compositor, which renders the video stream on a local display device.

Config JSON field description

Table 2-11 Field description–config_daisychain_detection_classification.json file

Field	Values/description
Input source	▪ `input-file`: The directory path to the video file. ▪ `rtsp-ip-port`: The address of the RTSP stream in *`rtsp://<ip>:<port>/<stream>`* format.
Models and labels	▪ `detection-model`: The path to the detection model. ▪ `detection-labels`: The path to the detection label. ▪ `classification-model`: The path to the classification model. ▪ `classification-labels`: The path to the classification label. ▪ `detection-constants`: The LiteRT detection model constants. ▪ `classification-constants`: The LiteRT classification model constants.

RELATED INFORMATION

"Image classification" on page 28
"Object detection" on page 33

2.1.10 Daisy chain detection and pose estimation

The **gst-ai-daisychain-detection-pose** application allows you to perform cascaded object detection and pose detection with a camera, file source, or an RTSP stream. The use cases involve detecting objects and estimating the body poses of the subject in an image or a video.

The figures show the application workflow, which receives the source, performs postprocessing, conducts inference on the AI hardware, and displays the results on the screen.

For information about the plugins used in the pipeline flow, see Pipeline flow.

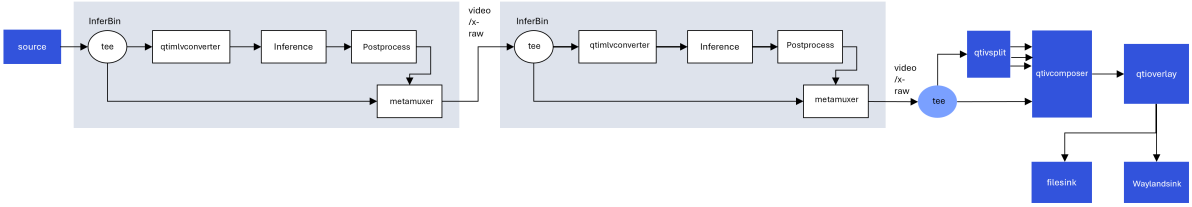


Figure 2-18 gst-ai-daisychain-detection-pose pipeline

Sample model and label files

Table 2-12 Sample model and label files for gst-ai-daisychain-detection-pose

Runtime	Model files	Label files
LiteRT	▪ Detection: *yolov8_det_quantized.tflite* ▪ Pose: *hrnet_pose_quantized.tflite*	▪ Detection: *yolov8.labels* ▪ Pose: *hrnet_pose.labels*

Prerequisites

- If not already done so, Download and install eSDK.
- Download model and label files.
- To access your host device, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is already enabled, you can skip this step.
- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.
- Enable the display:

  ```
  export XDG_RUNTIME_DIR=/dev/socket/weston && export
  WAYLAND_DISPLAY=wayland-1
  ```
- Rename the label files on the Linux host computer:

  ```
  cp /etc/labels/yolonas.labels /etc/labels/yolov8.labels
  ```
- Push the model files from the Linux host computer:

  ```
  scp <model_filename> root@<IP address of target device>:/etc/models
  ```

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

Run the application

For model and label files, see Sample model and label files.

To run the application, use the following syntax:

```
gst-ai-daisychain-detection-pose [OPTION?]
```

- Use a camera stream and see the output on the Wayland display:

  ```
  gst-ai-daisychain-detection-pose -c -d
  ```

- Use a file stream and save the output as a file:

  Push the `video.mp4` file to the `/etc/media` folder before running the command:

  ```
  gst-ai-daisychain-detection-pose --input-file=/etc/media/video.mp4 -o /etc/media/out.mp4
  ```

- Use an RTSP stream and see the output on the Wayland display:

  ```
  gst-ai-daisychain-detection-pose --rtsp-ip-port="rtsp://<ip>:port/<stream>" --display
  ```

To display the available help options, run the following command in the SSH shell:

```
gst-ai-daisychain-detection-pose -h
```

To stop the use case, use **CTRL + C**.

Expected output

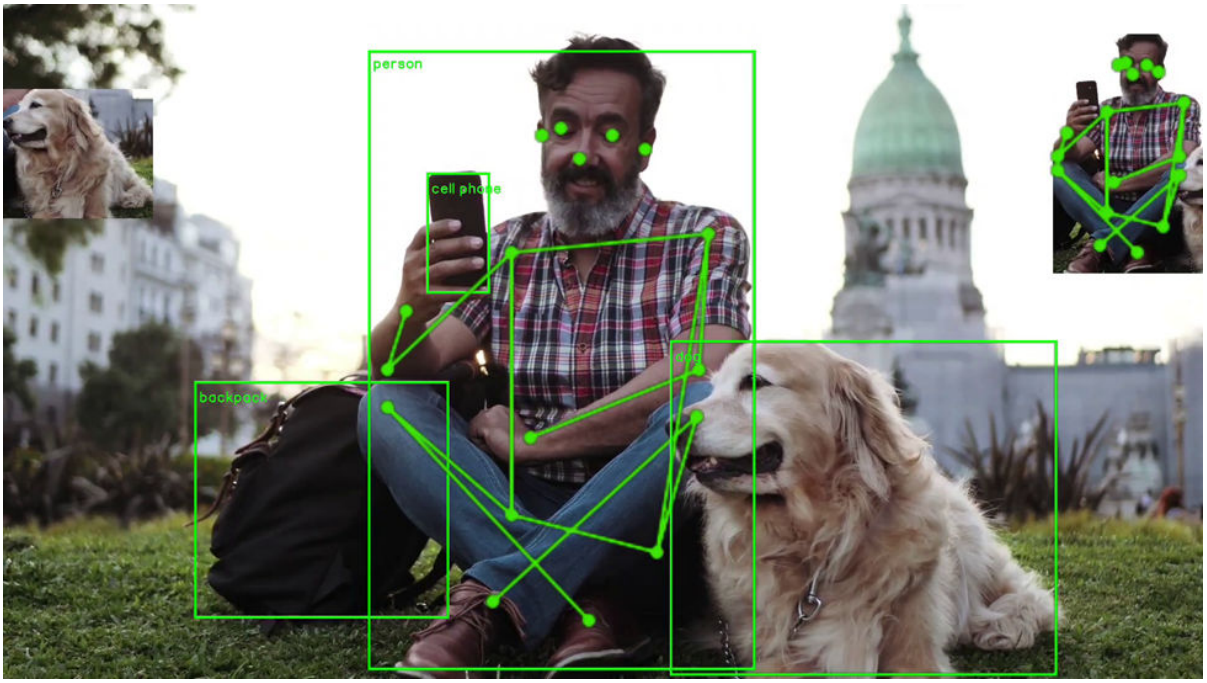


Figure 2-19 Expected output for gst-ai-daisychain-detection-pose application

Pipeline flow

The table lists the plugins used in the daisy chain detection and pose estimation pipeline:

Plugin	Description
Camera source:qtiqmmfsrc	▪ Captures the live stream from camera. ▪ Uses tee to split the stream for inferencing.
File source: filesrc	▪ Captures the video stream using filesrc, followed by qtdemux, which demultiplexes the stream. ▪ Uses tee to split the stream for inferencing.
RTSP source: rtspsrc	▪ Captures the RTSP stream using rtspsrc, followed by rtph264depay for video extraction. ▪ Uses tee to split the stream for inferencing.
qtimetamux	Multiplexes the stream.
h264parse	Parses the H.264 video.
v4l2h264dec	Decodes the H.264 video stream.
qtivsplit	Crops full frame into smaller frames based on the detected bounding boxes detected (maximum 4).
qtimlvconverter	Used by AI processing stream for preprocessing: 1. Receives the video stream on its sink pad. 2. Performs the following preprocessing on the stream data. This preprocessing is done when the model expects floating-point values as input. a. Color conversion b. Scaling (up or down) c. Normalization 3. Converts the preprocessed video stream to a tensor stream on its source pad. The tensor stream is used for inferencing in the later stages of the pipeline.
qtimltflite	1. After the inference runtime receives the tensor stream on its sink pad, it runs the inference. 2. Produces a tensor stream with the inference results on its source pad.
qtimlvdetection	Handles inference results from any object detection model. 1. Applies a threshold to the chosen number of results. 2. Loads the YOLOv8 module. 3. Generates results in the form of video frames with detection labels. 4. Produces video frames with only bounding boxes that can be cropped.
qtimlvpose	▪ Applies a threshold to the chosen number of results. ▪ Loads corresponding modules for various pose detection models. In the use case described in this section, qtimlvpose does the following: a. Loads the HRNet module. b. Produces results in the form of video frames with drawn poses. c. Sends the results to the sink pad of qtivcomposer for further processing or display.

Plugin	Description
qtivcomposer	1. Composes frames by combining the contents from its sink pads. 2. Pushes the GStreamer buffers containing the composed frames onto its source pad.
Waylandsink	1. Forwards the video stream received on its sink pad to Weston. 2. Weston renders the video stream on a local display.

RELATED INFORMATION

"Daisy chain detection and pose detection using Python" on page 202
"Object detection" on page 33

2.1.11 Monodepth from video

The **gst-ai-monodepth** application allows you to infer depth of a source feed from a live camera stream, file, or an RTSP stream.

The figure shows the pipeline, which captures feed from the source, preprocesses the video data, and runs inference using the AI hardware. For information about the plugins used in the pipeline, see Pipeline flow.

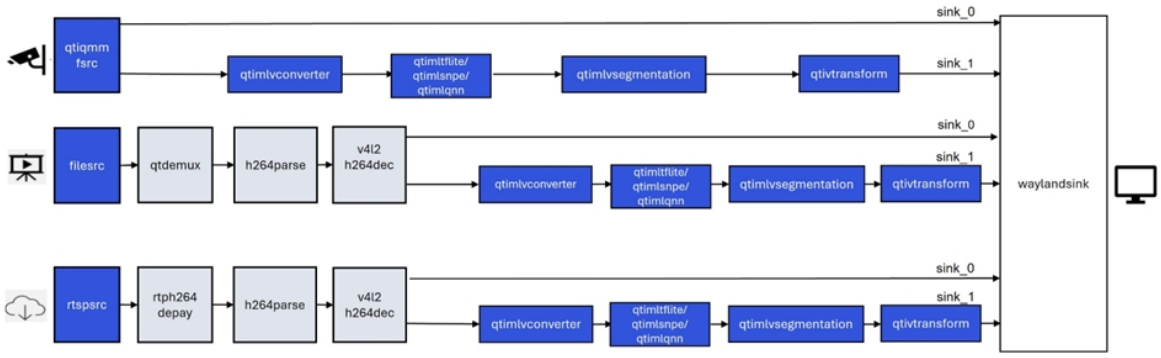


Figure 2-20 gst-ai-monodepth pipeline

Sample model and label files

Table 2-13 Sample model and label files for gst-ai-monodepth

Runtime	Model files	Label files
Qualcomm Neural Processing SDK	`midasv2.dlc`	`monodepth.labels`
LiteRT	`midas_quantized.tflite`	
Qualcomm AI Engine direct	`midas_quantized.bin`	

Prerequisites

- If not already done so, Download and install eSDK.
- Download model and label files.

  The application supports the Qualcomm Neural Processing SDK, Qualcomm AI Engine direct, and LiteRT models.

- To access your host device, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is already enabled, you can skip this step.
- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.
- Enable the display:

  ```
  export XDG_RUNTIME_DIR=/dev/socket/weston && export
  WAYLAND_DISPLAY=wayland-1
  ```

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

Run the application

The sample application uses the `/etc/configs/config_monodepth.json` file to read the input parameters.

To create your own config JSON file, use config_monodepth.json as a reference.

1. Use the following format of the `config_monodepth.json` file:

   ```
   {
     "file-path": "<input video path>",
     "ml-framework": "<snpe, tflite, or qnn framework>",
     "model": "<path-to-model-file>",
     "labels": "<path-to-label-file>",
     "constants": "<model-constants-for-quantized-LiteRT-model>",
     "runtime": "<dsp, gpu, or cpu runtime>"
   }
   ```

   NOTE Update the config JSON file based on the model, input stream, and other properties. For more information, see Config JSON field description.

   For example, run the application using the LiteRT model and DSP runtime, with input from a video file, and custom model and label paths:

   ```
   {
       "file-path": "/etc/media/video.mp4",
       "ml-framework": "tflite",
       "model": "/etc/models/midas_quantized.tflite",
       "labels": "/etc/labels/monodepth.labels",
       "constants": "Midas,q-offsets=<0.0>,q-scales=<4.716535568237305>;",
       "runtime": "dsp"
     }
   ```

2. Run the gst-ai-monodepth application:

   ```
   gst-ai-monodepth --config-file=/etc/configs/config_monodepth.json
   ```

To display the available help options, run the following command in the SSH shell:

```
gst-ai-monodepth -h
```

To stop the use case, use **CTRL + C**.

Expected output

The overlaid model output stream is shown side by side with the live feed.

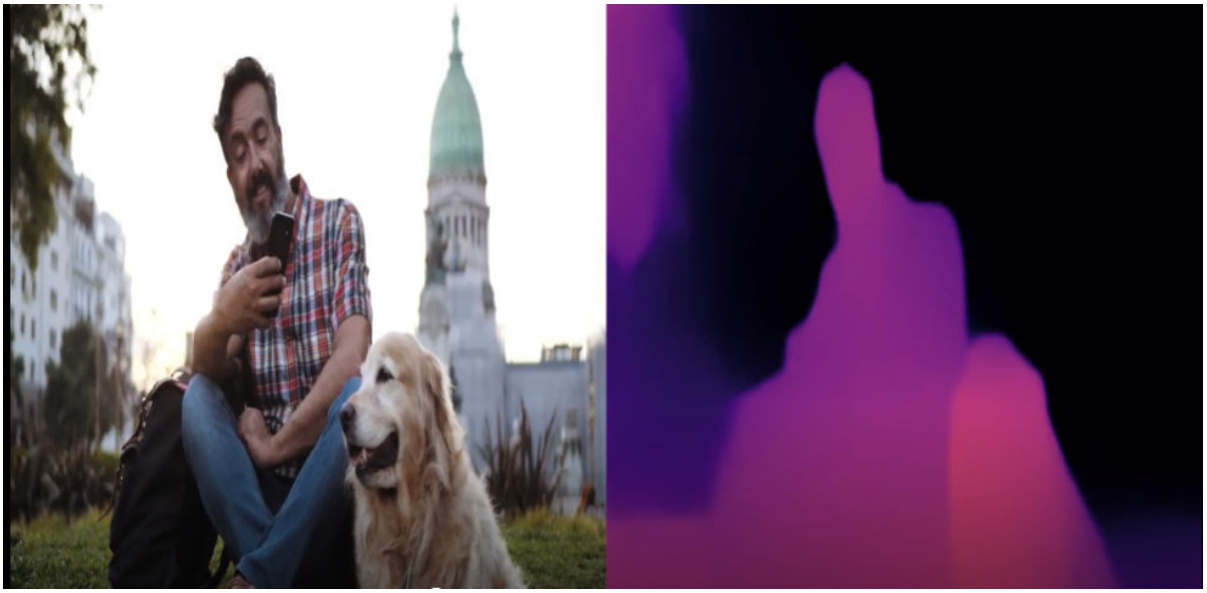


Figure 2-21 Expected output for gst-ai-monodepth application

Pipeline flow

The table lists the plugins used in the mono depth pipeline:

Plugin	Description
Camera source:qtiqmmfsrc	▪ Captures the live stream from camera. ▪ Uses tee to split the stream for inferencing.
File source: filesrc	▪ Captures the video stream using filesrc, followed by qtdemux, which demultiplexes the stream. ▪ Uses tee to split the stream for inferencing.
RTSP source: rtspsrc	▪ Captures the RTSP stream using rtspsrc, followed by rtph264depay for video extraction. ▪ Uses tee to split the stream for inferencing.
h264parse	Parses the H.264 video.
v4l2h264dec	Decodes the video.
qtimlvconverter	Used by AI processing stream for preprocessing: 1. Receives the video stream on its sink pad. 2. Performs the following preprocessing on the stream data. This preprocessing is done when the model expects floating-point values as input. a. Color conversion b. Scaling (up or down) c. Normalization 3. Converts the preprocessed video stream to a tensor stream on its source pad. The tensor stream is used for inferencing in the later stages of the pipeline.

Plugin	Description
Inferencing plugins: qtimlsnpe, qtimltflite, and qtimlqnn	Uses the Midasv2 model for monodepth. 1. The inference runtime receives the tensor stream on its sink pad. 2. The runtime runs the inference. 3. Produces a tensor stream with the inference results on its source pad. The postprocessing plugin for processing the inference comes from the Midasv2 model.
qtimlvsegmentation	Converts the inference tensors that it receives on its sink pad into video formats that the multimedia plugins can use for further processing.
qtivtransform	Converts the buffers on its source pad. These buffers are for composition on Waylandsink.
Waylandsink	1. Waylandsink submits the video stream received on its sink pad to Weston. 2. Weston renders the video stream on a local display.

Config JSON field description

Table 2-14 Field description–config_monodepth.json file

Field	Values/description
ml-framework	Enable and use one of the following models: ▪ `snpe`: Qualcomm Neural Processing SDK ▪ `tflite`: LiteRT ▪ `qnn`: Qualcomm AI Engine direct
runtime	Enable and use one of the following runtimes: ▪ `cpu` ▪ `gpu` ▪ `dsp`
Input source	Enable and use one of the following input sources: ▪ `camera`: Primary (0) or secondary (1). ▪ `file-path`: The directory path to the video file. ▪ `rtsp-ip-port`: The address of the RTSP stream: *`rtsp://<ip>:<port>/<stream>`*

RELATED INFORMATION

"Image segmentation" on page 42

2.1.12 Video super-resolution

The **gst-ai-superresolution** application allows you to generate high resolution video frames from low-resolution input.

NOTE This application isn't supported on QCS9075.

The figure shows the pipeline, which accepts a video stream from a file source as input, processes it through the super resolution module using LiteRT, and displays the output.

For information about the plugins used in the pipeline, see Pipeline flow.

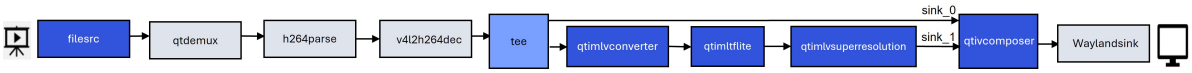


Figure 2-22 gst-ai-superresolution pipeline (Wayland display)

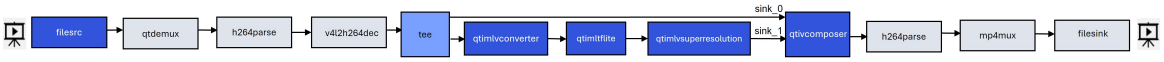


Figure 2-23 gst-ai-superresolution pipeline (file sink)

Sample model files

Table 2-15 Sample model for gst-ai-superresolution

Runtime	Model files
LiteRT	*quicksrnetsmall_quantized.tflite*

Prerequisites

- Download model and label files.
- To access your host device, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is already enabled, you can skip this step.
- Start two SSH shells–one to enable Weston and the other to run the use cases:

  ```
  ssh root@<ip-addr of the target device>
  ```
- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.
- Enable the display:

  ```
  export XDG_RUNTIME_DIR=/dev/socket/weston && export
  WAYLAND_DISPLAY=wayland-1
  ```
- Push the model files from the Linux host computer:

  ```
  scp <model_filename> root@<IP address of target device>:/etc/models
  ```
- Note that the Download model and label files downloads the sample `video.mp4` video to the `/etc/media` directory. If you are using a custom video, then ensure that you push the video to `/etc/media` and update the file path in the application config. JSON file.

  NOTE The expected resolution of the video file is 128 × 128.

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

Run the application

- Display the output with Wayland display:

  ```
  gst-ai-superresolution --input-file=/etc/media/video.mp4 --display
  ```

- Save the output to a file using filesink:

  ```
  gst-ai-superresolution --input-file=/etc/media/video.mp4 --output-
  file=/etc/media/out.mp4
  ```

- Run a model with constants using LiteRT:

  ```
  gst-ai-superresolution --input-file=/etc/media/video.mp4 --model=/etc/
  models/quicksrnetsmall_quantized.tflite --constants="srnet,q-
  offsets=<0.0>,q-scales=<1.0>;"
  ```

  NOTE The values for the `q-scales` and `q-offsets` constants are `<1.0>` and `<0.0>` respectively.

To display the available help options, run the following command in the SSH shell:

```
gst-ai-superresolution -h
```

To stop the use case, use **CTRL + C**.

Expected output

The output is displayed on an HDMI monitor.

Figure 2-24 Expected output for VSR

Pipeline flow

The table lists the plugins used in the video super resolution pipeline:

Plugin	Description
filesrc	Captures the video stream and uses tee to split the stream for inferencing.
qtimlvconverter	Used by AI processing stream for preprocessing: 1. Receives the video stream on its sink pad. 2. Performs the following preprocessing on the stream data. This preprocessing is done when the model expects floating-point values as input. a. Color conversion b. Scaling (up or down) c. Normalization 3. Converts the preprocessed video stream to a tensor stream on its source pad. The tensor stream is used for inferencing in the later stages of the pipeline.
qtimltflite	Runs on LiteRT and uses the `quicksrnetsmall_quantized` model.

Plugin	Description
	1. The inference runtime receives the tensor stream on its sink pad. 2. The runtime runs the inference. 3. Produces a tensor stream with the inference results on its source pad.
qtimlvsuperresolution	Handles inference results from any super resolution mode. 1. Loads SRNet module. 2. Produces results as video frames. 3. Sends them to the sink pad of qtivcomposer. For the INT8 model, add the following constants: ▪ q-offsets ▪ q-scales
qtivcomposer	1. Composes frames with contents from its sink pads. 2. Pushes the GStreamer buffers containing these composed frames to its source pad.
Waylandsink	1. Waylandsink submits the video stream received on its sink pad to Weston. 2. Weston renders the video stream on a local display.

RELATED INFORMATION

"Video super resolution and display with LiteRT" on page 244

2.1.13 Multistream inference

The **gst-ai-multistream-inference** application shows AI inference (object detection and classification) on up to 32 input streams coming from camera, file, or RTSP stream.

The figure shows the pipeline, which takes several input streams, does the preprocessing and AI inferencing stream, combines the streams, and merges them all into a single video output.

NOTE The maximum number of input streams supported on each SoC are follows:

- QCS6490–8
- QCS8275–16
- QCS9075–32

The output is displayed on an HDMI display, saved as an H.264 encoded MP4 file, or converted into an RTSP stream.

For information about the plugins used in this pipeline, see Pipeline flow.

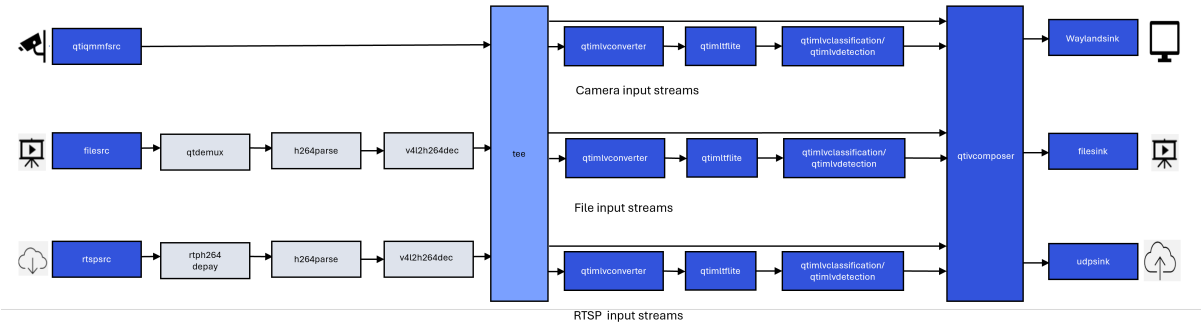


Figure 2-25 gst-ai-multistream-inference pipeline

Sample model and label files

Table 2-16 Sample model and label files for gst-ai-multistream-inference

Runtime	Model files	Label files
LiteRT	▪ *inception_v3_quantized.tflite* ▪ *yolov8_det_quantized.tflite*	▪ *yolov8.labels* ▪ *Classification.labels*

Prerequisites

- If not already done so, Download and install eSDK.
- Download model and label files.
- To access your host device, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is already enabled, you can skip this step.
- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.
- Enable the display:

```
export XDG_RUNTIME_DIR=/dev/socket/weston && export
WAYLAND_DISPLAY=wayland-1
```

- Push the video files from the Linux host computer:

```
scp <filename>.mp4 root@<IP address of target device>:/etc/media
```

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

Run the application

The sample application uses the `/etc/configs/config-multistream-inference.json` file to read the input parameters.

NOTE In the `config-multistream-inference.json` file, you can replace `model-detection.tflite` with `yolov8_det_quantized.tflite` and `detection.labels` with `yolov8.labels`.

To create your own config JSON file, use config-multistream-inference.json as a reference.

1. Use the following format of the `config-multistream-inference.json` file:

```
{
  "input-file-path": [
    "<input file 1>",
    "<input file 2>"
  ],
  "input-rtsp-path": [
    "<rtsp stream 1>",
    "<rtsp stream 2 >"
  ],
  "input-type": "<h264 or h265>"
  "model": "<path to model file>",
```

```
  "labels": "<path to label file>",
  "constants" : "<constants for LiteRT model>",
  "output-display": <0 or 1>,
  "output-file-path": "<output file path>",
  "output-ip-address": "<IP address of test kit>",
  "output-port-number": "<port number over which rtsp stream can be
listened>",
  "use-case": <0 or 1>
}
```

NOTE Update the config JSON file based on the model, input stream, and other properties. For more information, see Config JSON field description.

For example, run the application for object detection on 16 H.264 file inputs and view the output on a Wayland display:

```
{
  "input-file-path":
    [
      "/etc/media/video1.mp4",
      "/etc/media/video2.mp4",
      "/etc/media/video3.mp4",
      "/etc/media/video4.mp4",
      "/etc/media/video5.mp4",
      "/etc/media/video6.mp4",
      "/etc/media/video7.mp4",
      "/etc/media/video8.mp4",
      "/etc/media/video9.mp4",
      "/etc/media/video10.mp4",
      "/etc/media/video11.mp4",
      "/etc/media/video12.mp4",
      "/etc/media/video13.mp4",
      "/etc/media/video14.mp4",
      "/etc/media/video15.mp4",
      "/etc/media/video16.mp4"
    ],
  "model": "/etc/models/yolov8_det_quantized.tflite",
  "labels": "/etc/labels/yolov8.labels",
  "constants": "YOLOv8,q-offsets=<21.0, 0.0, 0.0>,q-
scales=<3.0546178817749023, 0.003793874057009816, 1.0>;",
  "input-type": "h264",
  "output-display": 1,
  "use-case": 0
}
```

2. Run the gst-ai-multistream-inference application:

```
gst-ai-multistream-inference --config-file=/etc/configs/config-multistream-
inference.json
```

To display the available help options, run the following commands in the SSH shell:

```
gst-ai-multistream-inference -h
```

To stop the use case, use **CTRL + C**.

Expected output

Figure 2-26 Expected output for gst-ai-multistream-inference

Pipeline flow

The table lists the plugins used in the multistream inference pipeline:

Plugin	Description
Camera source:qtiqmmfsrc	▪ Captures the live stream from camera. ▪ Uses tee to split the stream for inferencing.
File source: filesrc	▪ Captures the video stream using filesrc, followed by qtdemux, which demultiplexes the stream. ▪ Uses tee to split the stream for inferencing.
RTSP source: rtspsrc	▪ Captures the RTSP stream using rtspsrc, followed by rtph264depay for video extraction. ▪ Uses tee to split the stream for inferencing.
h264parse	Parses the H.264 video.
v4l2h264dec	Decodes the video.

Plugin	Description
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs the following preprocessing on the stream data. This preprocessing is done when the model expects floating-point values as input. a. Color conversion b. Scaling (up or down) c. Normalization The tensor stream is used for inferencing in the later stages of the pipeline.
qtimltflite	It runs on LiteRT and uses a quantized YOLOv8 model for object detection and a quantized Inception v3 model for classification. 1. After the inference runtime receives the tensor stream on its sink pad, it runs the inference. 2. Produces a tensor stream with the inference results on its source pad.
Postprocessing plugins	Handles the inference results from any object detection, classification, pose detection, and segmentation model. ■ qtimlvdetection: a. Applies a threshold to the chosen number of results. b. Loads the YOLO-NAS module. c. Produces video frames with only bounding boxes that can be overlaid on objects, sending them to the sink pad of the qtivcomposer. ■ qtimlvclassification a. Applies the threshold to the chosen number of results. b. Loads the MobileNet module. c. Produces results as video frames with classification labels, sending them to the sink pad of the qtivcomposer.
qtivcomposer	1. Composes frames with contents from its sink pads. 2. Pushes the GStreamer buffers containing these composed frames to its source pad.
Waylandsink	1. Waylandsink submits the video stream received on its sink pad to Weston. 2. Weston renders the video stream on a local display.
Filesink	1. Receives the video stream on its sink pad. 2. Saves the stream as a H.264-encoded MP4 file.
qtirtspbin	1. Serves as a network sink. 2. Transmits UDP packets to the network.

Config JSON field description

Table 2-17 Field description–gst-ai-multistream-inference file

Field	Values/description
Input source	Use one of the following input sources: ■ `num-camera`: The number of inputs from the camera. ■ `camera`: The input camera if `num-camera=1`. ■ `input-file-path`: The directory path to the video file. ■ `input-rtsp-path`: The address of the RTSP stream: *rtsp://<ip>:<port>/<stream>*
input-type	The video encoding type for file and RTSP input:

Table 2-17 Field description–gst-ai-multistream-inference file (cont.)

Field	Values/description
	▪ H.264 ▪ H.265
Output	Use one of the following outputs: ▪ `output-file-path`: The directory path to save the output file. ▪ `output-ip-address`: The IP address of the device on which the RTSP stream can be played. ▪ `output-port-number`: The port number of the device on which the RTSP stream can be played. ▪ `output-display`: The connected display device for preview. Select 1 to enable this option.

Known issues

Segmentation fault observed with 32 streams at end of stream (EOS).

RELATED INFORMATION

"Image classification" on page 28
"Object detection" on page 33

2.1.14 Multistream batch inference

The **gst-ai-multistream-batch-inference** application shows batched AI inference (object detection and segmentation) on up to 24 input streams from video files.

NOTE The maximum number of input streams supported on each SoC are follows:

- QCS6490–8
- QCS8275–4
- QCS9075–4

The figure shows the pipeline, which takes several input streams, does the preprocessing and AI inferencing stream on each video stream, combines the streams with inference, and merges them into a single video output.

The output is displayed either on an HDMI display or saved as an H.264 encoded MP4 file.

For information about the plugins used in this pipeline, see Pipeline flow.

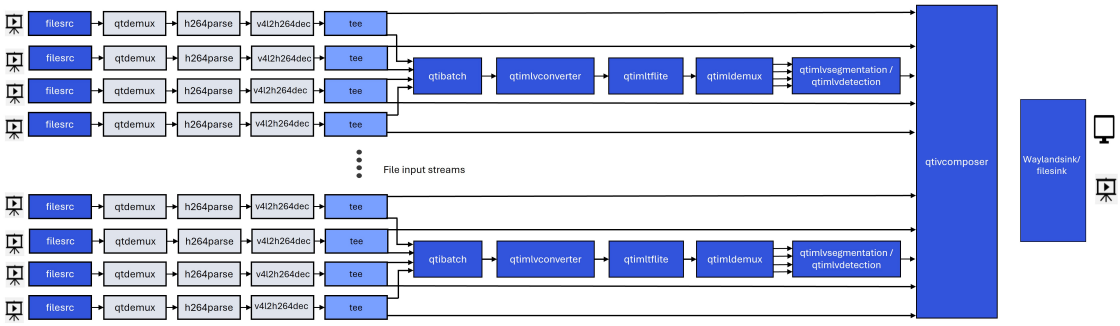


Figure 2-27 gst-ai-multistream-batch-inference pipeline

Sample model and label files

Table 2-18 Sample model and label files for gst-ai-multistream-batch-inference

Runtime	Model files	Label files
LiteRT	▪ Segmentation: *deeplabv3_plus_mobilenet_quantized.tflite* ▪ Object detection: *yolov8_det_quantized.tflite*	▪ Segmentation: *deeplabv3_resnet50.labels* ▪ Object detection: *yolov8.labels*

Prerequisites

- If not already done so, Download and install eSDK.
- To run the application, push the model files.

  The application supports the `deeplabv3_plus_mobilenet_quantized.tflite` and `yolov8_det_quantized.tflite` models for batching.

  - To generate the batching model, see Download model and label files.
  - For model files, see Sample model and label files.

  NOTE The gst-ai-multistream-batch-inference application uses the `batch-size=4` model. To change the batch size of the model, update the `export` command:

  ```
  python -m qai_hub_models.models.<Model_Name>.export --quantize
  w8a8 --batch-size <N> --device "QCS6490 (Proxy)"
  ```

- To access your host device, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is already enabled, you can skip this step.

- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.
- Enable the display:

  ```
  export XDG_RUNTIME_DIR=/dev/socket/weston && export
  WAYLAND_DISPLAY=wayland-1
  ```

- Push the model files from the Linux host computer:

  ```
  scp <model_filename> root@<IP address of target device>:/etc/models
  ```

- Export the `cdsp1` library path:

  ```
  export ADSP_LIBRARY_PATH="/usr/lib/dsp/cdsp;/usr/lib/dsp/
  cdsp1;$ADSP_LIBRARY_PATH"
  ```

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

Run the application

The sample application uses the `/etc/configs/config-multistream-batch-inference.json` file to read the input parameters

To create your own config JSON file, use config-multistream-batch-inference.json as a reference.

1. Use the following format of the `config-multistream-batch-inference.json` file:

   For 16 and 24 streams, add the required elements in the `pipeline-info` parameter. The `id` parameter takes the values from 0 to 5 for each added batch.

```
{
“output-type”: “wayland or filesink”,
"out-file":"<Path to output file if output-type is filesink>",
 “pipeline-info”:[
    {
  "id": "<Batch id takes value from 0 to 5>",
  “Input type”: “<Input source type like file>”,
  “input-file-path”: [
    {
    “<stream[i]: Comma separated input file/RTSP path>”
    }
   ],
"mlframework": "<qtimltflite>",
"model-path": "<path-to-model-file>",
"labels": "<path-to-label-file>",
"constants": "<model-constants-for-quantized-LiteRT-model>",
"post processing plugin": "< qtimlvdetection or qtimlvsegmentation>"
    }
  ]
}
```

   NOTE Update the config JSON file based on the model, input stream, and other properties. For more information, see Config JSON field description.

   In the `config-multistream-batch-inference.json` file, with the batch size as 4, you can replace the following:

   - `model-batch-4.tflite` with `yolov8_det_quantized.tflite`
   - `detection.labels` with `yolov8.labels`
   - `constants` with`"yolov8,q-offsets=<24.0, 0.0, 0.0>,q-scales=<3.146998405456543, 0.003778052283450961, 1.0>;"`

   For example, run the application using the LiteRT model, video file source, label paths, and the qtimlvdetection postprocessing plugin along with Wayland output.

```
{
  "output-type":"wayland",
  "pipeline-info":[
    {
      "id":0,
      "input-type":"file",
      "input-file-path":[
        {
            "stream-0":"/etc/media/video.mp4",
            "stream-1":"/etc/media/video.mp4",
```

```
                "stream-2":"/etc/media/video.mp4",
                "stream-3":"/etc/media/video.mp4"
            }
        ],
        "mlframework":"qtimltflite",
        "model-path":"/etc/models/yolov8_det_quantized.tflite",
        "labels-path":"/etc/labels/yolov8.labels",
        "constants":"yolov8,q-offsets=<24.0, 0.0, 0.0>,q-
scales=<3.146998405456543, 0.003778052283450961, 1.0>;",
        "post-process-plugin": "qtimlvdetection"
      }
    ]
}
```

2. Run the gst-ai-multistream-batch-inference application:

```
gst-ai-multistream-batch-inference --config-file=/etc/configs/config-
multistream-batch-inference.json
```

To display the available help options, run the following command in the SSH shell:

```
gst-ai-multistream-batch-inference -h
```

To stop the use case, use **CTRL + C**.

Expected output

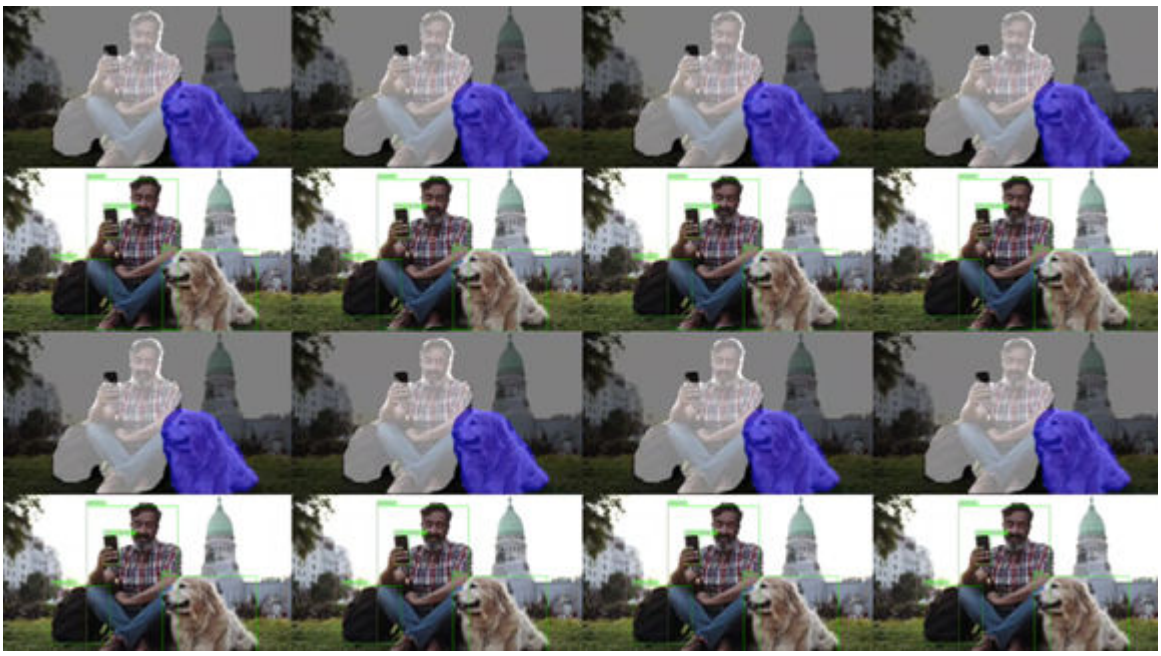


Figure 2-28 Expected output for gst-ai-multistream-batch-inference–Preview

Pipeline flow

The table lists the plugins used in the multistream batch inference pipeline:

Plugin	Description
File source: filesrc	▪ Captures the video stream using filesrc, followed by qtdemux, which demultiplexes the stream. ▪ Uses tee to split the stream for inferencing.
h264parse	Parses the H.264 video.
v4l2h264dec	Decodes the video.
qtibatch	▪ Reads input from the streams on its sink pad. ▪ Batches the streams for preprocessing.
qtimlvconverter	1. Receives the batched video streams on its sink pad. 2. Performs the following preprocessing on the stream data. This preprocessing is done when the model expects floating-point values as input. a. Color conversion b. Scaling (up or down) c. Normalization The tensor stream is used for inferencing in the later stages of the pipeline.
qtimltflite	1. After the inference runtime receives the tensor stream on its sink pad, it runs the inference. 2. Produces a tensor stream with the inference results on its source pad.
qtimldemux	1. Demultiplexes the batched output. 2. Splits the output corresponding to the input streams.
Postprocessing plugins	qtimlvsegmentation and qtimlvdetection: Convert the inference tensors that are received on the sink pad into video formats the multimedia plugins can use for further processing.
qtivcomposer	1. Composes frames with contents from its sink pads. 2. Pushes the GStreamer buffers containing these composed frames to its source pad.
Waylandsink	1. Waylandsink submits the video stream received on its sink pad to Weston. 2. Weston renders the video stream on a local display.
Filesink	Takes the video stream that it receives on its sink pad and saves it as a H.264-encoded MP4 file.

Config JSON field description

Table 2-19 Field description–gst-ai-multistream-batch-inference file

Field	Values/description
output type	Use one of the following output type: ▪ `Wayland`:Displays output on Weston. ▪ `filesink`: Encodes the output in a video file.
out-file	▪ The file path to save the output file.
pipeline -info	Provides the pipeline information: ▪ `Stream id`: Ranges from 0 to 5 ▪ `Input-type`: The input source file. ▪ `Input-file-path`: The array of the input file path.

Table 2-19 Field description–gst-ai-multistream-batch-inference file (cont.)

Field	Values/description
mlframework	Uses the inference plugin such as qtimltflite.
model-path	The path to the model file.
labels-path	The path to the labels file.
constants	The constant of the LiteRT model.

Known issues

Frame freeze is observed some time after starting the application.

RELATED INFORMATION

"Object detection" on page 33
"Image segmentation" on page 42

2.1.15 AI smart codec

The **gst-ai-smartcodec-example** application reduces network bandwidth and storage resources for input from a camera or a file source.

Sample model and label files

Table 2-20 Sample model and label files for gst-ai-smartcodec-example

Runtime	Model files	Label files
LiteRT	*YOLOv8-Detection-Quantized.tflite*	*coco_labels.txt*

Prerequisites

- If not already done so, Download and install eSDK.
- Download model and label files.
- To access your host computer, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is already enabled, you can skip this step.
- Push the model files from the Linux host computer.

  ```
  scp <model_filename> root@<IP address of target device>:/etc/models
  ```
- Note that the Download model and label files downloads the sample `video.mp4` video to the `/etc/media` directory. If you are using a custom video, then ensure that you push the video to `/etc/media` and update the file path in the application config.JSON file.
- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.
- Enable the display:

  ```
  export XDG_RUNTIME_DIR=/dev/socket/weston && export
  WAYLAND_DISPLAY=wayland-1
  ```

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

Run the application

For model and label files, see Sample model and label files.

- Run the application with a camera source:

```
gst-ai-smartcodec-example -w 640 -h 480 -o output.mp4 -m /etc/models/
YOLOv8-Detection-Quantized.tflite -l /etc/labels/coco_labels.txt
```

- Run the application with a file source:

```
gst-ai-smartcodec-example -i /etc/media/video.mp4 -o output.mp4 -m /etc/
models/YOLOv8-Detection-Quantized.tflite -l /etc/labels/coco_labels.txt
```

To display the available help options, run the following commands in the SSH shell:

```
gst-ai-smartcodec-example --help
```

To stop the use case, use **CTRL + C**.

Expected output

The application generates the output as an encoded MP4 file.

RELATED INFORMATION

"Smart codec" on page 147

2.1.16 Face detection

The **gst-ai-face-detection** application collects the live video input from a camera, file, or an RTSP stream and uses the Qualcomm AI Engine direct and LiteRT face detection models to produce a preview with the overlaid AI model output on the HDMI display.

The figure shows the pipeline, which receives the input, performs preprocessing, runs inferences on AI hardware, and displays the results on the screen.

For information about the plugins used in the pipeline flow, see Pipeline flow.

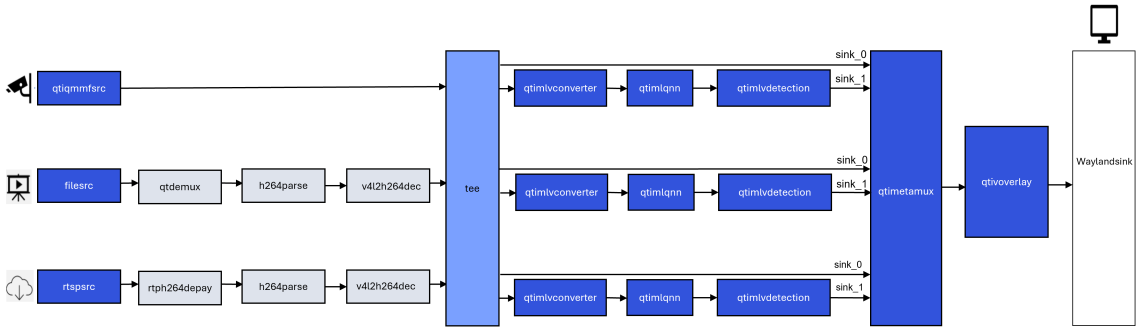


Figure 2-29 gst-ai-face-detection pipeline

Sample model and label files

Table 2-21 Sample model and label files for gst-ai-face-detection

Runtime	Model files	Label files
LiteRT	*face_det_lite_quantized.tflite*	*face_detection.labels*
Qualcomm AI Engine direct	*face_det_lite_quantized.bin*	

Prerequisites

- If not already done so, Download and install eSDK.
- Download model and label files.
- To access your host computer, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is already enabled, you can skip this step.
- Push the model files from the Linux host computer.

  ```
  scp <model_filename> root@<IP address of target device>:/etc/models
  ```
- Note that the Download model and label files downloads the sample `video.mp4` video to the `/etc/media` directory. If you are using a custom video, then ensure that you push the video to `/etc/media` and update the file path in the application config.JSON file.
- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.
- Enable the display:

  ```
  export XDG_RUNTIME_DIR=/dev/socket/weston && export
  WAYLAND_DISPLAY=wayland-1
  ```

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

Run the application

The sample application uses the `/etc/configs/config_face_detection.json` file to read the input parameters.

To create your own config JSON file, use config_face_detection.json as a reference.

1. Use the following format of the `config_face_detection.json` file:

   ```
   {
   "file-path": "<input-video-path>",
   "ml-framework": "<tflite or qnn framework>",
   "model": "<path-to-model-file",
   "labels": "<path-to-label-file",
   "constants": "<model-constants-for-LiteRT-model>",
   "threshold": <post-processing threshold, integer value from 1-100>,
   ```

```
"runtime": ""<cpu, gpu or dsp runtime>"
}
```

NOTE Update the config JSON file based on the model, input stream, and other properties. For more information, see Config JSON field description.

For example, run the application using LiteRT, input video file, custom model, custom label file, DSP runtime, and custom threshold:

```
{
"file-path": "/etc/media/video.mp4", "ml-framework": "tflite",
"model":"/etc/models/face_det_lite_quantized.tflite",
"labels": "/etc/labels/face_detection.labels",
"constants": "qfd,q-offsets=<204.0, 12.0, 102.0>,q-
scales=<0.0365355908870697,0.23025865852832794, 0.10316766798496246>;",
"threshold": 51,
"runtime": "dsp"
}
```

2. Run the gst-ai-face-detection application:

```
gst-ai-face-detection --config-file=/etc/configs/config_face_detection.json
```

To display the available help options, run the following commands in the SSH shell:

```
gst-ai-face-detection -h
```

To stop the use case, use **CTRL + C**.

Pipeline flow

The table lists the plugins used in the face detection pipeline:

Plugin	Description
Camera source:qtiqmmfsrc	▪ Captures the live stream from camera. ▪ Uses tee to split the stream for inferencing.
RTSP source: rtspsrc	▪ Captures the RTSP stream using rtspsrc, followed by rtph264depay for video extraction. ▪ Uses tee to split the stream for inferencing.
h264parse	Parses the H.264 video.
v4l2h264dec	Decodes the video.
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs the following preprocessing on the stream data. This preprocessing is done when the model expects floating-point values as input. a. Color conversion b. Scaling (up or down) c. Normalization 3. Converts the preprocessed video stream to a tensor stream on its source pad. The tensor stream is used for inferencing in the later stages of the pipeline.
▪ qtimltflite ▪ qtimlqnn	Acts as the inferencing plugin.

Plugin	Description
	1. After the inference runtime receives the tensor stream on its sink pad, the plugin runs the inference. 2. Produces a tensor stream with the inference results on its source pad.
qtimlvdetection	▪ Handles inference results from any face detection model. ▪ Applies a threshold to the chosen number of results.
qtimetamux	Receives string-based postprocessing output text with video frame and multiplexes it.
qtivoverlay	1. Receives the multiplexed stream. 2. Overlays the bounding boxes on the stream.
Waylandsink	1. Waylandsink submits the video stream received on its sink pad to Weston. 2. Weston renders the video stream on a local display.

Config JSON field description

Table 2-22 Field description–gst-ai-face-detection file

Field	Values/description
ml-framework	Use one of the following models: ▪ `tflite`: LiteRT ▪ `qnn`: Qualcomm AI Engine direct
runtime	Use one of the following runtimes: ▪ `cpu` ▪ `gpu` ▪ `dsp`
Input source	Use one of the following input sources: ▪ `camera`: Primary (0) or secondary (1). ▪ `file-path`: The directory path to the video file. ▪ `rtsp-ip-port`: The address of the RTSP stream: *rtsp://<ip>*

RELATED INFORMATION

"Object detection" on page 33

2.1.17 Face recognition

The **gst-ai-face-recognition** application collects the live video input from a camera or an RTSP stream and shares this input for face detection, facial landmarking and face recognition. It uses the `face_det_quantized` models for face detection, `facemap_3dmm_quantized` model for facial landmarking, and `face_attrib_net_quantized` model for face recognition labels.

The result is a preview of the overlaid AI model on the HDMI display.

The figure shows the pipeline, which receives the input, performs preprocessing, runs inferences on AI hardware, and displays the results on the screen.

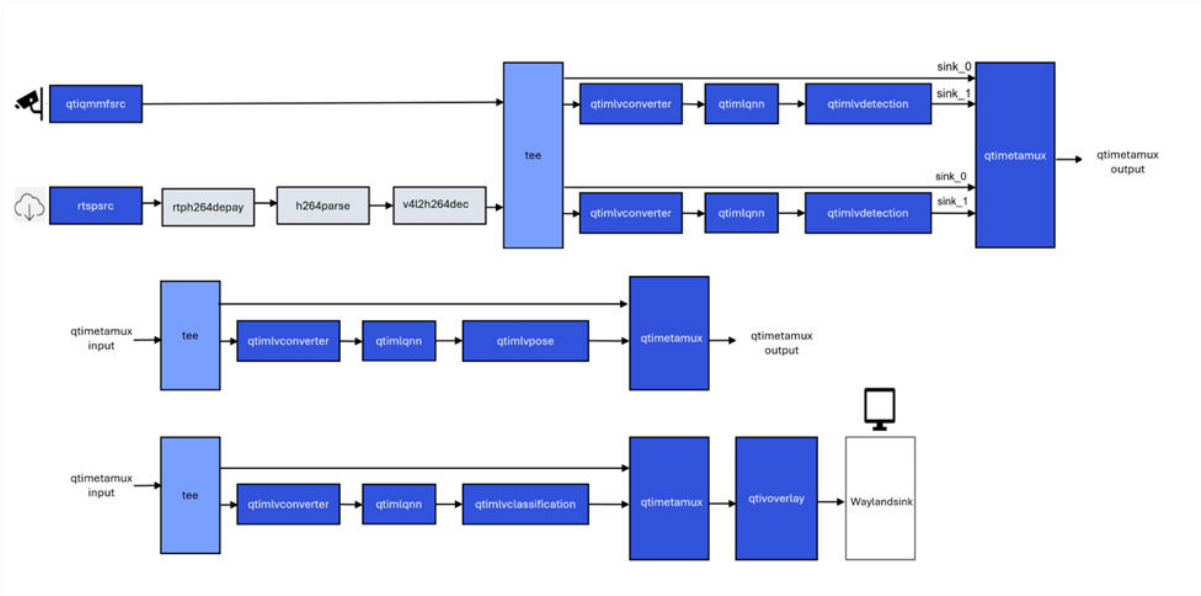


Figure 2-30 gst-ai-face-recognition pipeline

For information about the plugins used in the pipeline flow, see Pipeline flow.

Sample model and label files

Table 2-23 Sample model and label files for gst-ai-face-detection

Runtime	Model files	Label files
Qualcomm AI Engine direct and LiteRT	▪ *face_det_lite_quantized.tflite* ▪ *facemap_3dmm_quantized.tflite* ▪ *face_attrib_net_quantized.tflite* ▪ *face_det_lite_quantized.bin* ▪ *facemap_3dmm_quantized.bin* ▪ *face_attrib_net_quantized.bin*	▪ *face_detection.labels* ▪ *face_recognition.labels* ▪ *face_landmark.labels*

Prerequisites

- If not already done so, Download and install eSDK.
- Download model and label files.
- To access your host computer, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is already enabled, you can skip this step.
- Push the model files from the Linux host computer.

  ```
  scp <model_filename> root@<IP address of target device>:/etc/models
  ```
- Note that the Download model and label files downloads the sample `video.mp4` video to the `/etc/media` directory. If you are using a custom video, then ensure that you push the video to `/etc/media` and update the file path in the application config.JSON file.

- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.
- Enable the display:

```
export XDG_RUNTIME_DIR=/dev/socket/weston && export
WAYLAND_DISPLAY=wayland-1
```

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

- Download the following Qualcomm AI Engine direct models:
  - face_det_quantized
  - face_attrib_net_quantized
- Push these models to the `/etc/models` directory on the target device.

```
scp face_det_lite_quantized.bin root@<IP address of target device>:/etc/
models
```

```
scp face_attrib_net_quantized.bin root@<IP address of target device>:/etc/
models
```

Register a face for facial recognition

Before running the gst-ai-face-recognition application, you can register a face for secure verification and authentication.

1. To register a face, use the following gst-pipeline on the target device shell:

```
gst-pipeline-app -e \
qtimlvconverter name=stage_01_preproc mode=image-batch-non-cumulative \
qtimlqnn name=stage_01_inference model=/etc/models/
face_det_lite_quantized.bin backend=/usr/lib/libQnnHtp.so \
qtimlvdetection name=stage_01_postproc stabilization=false threshold=40.0
results=4 module=qfd labels=/etc/labels/face_detection.labels \
qtimlvconverter name=stage_03_preproc mode=roi-batch-cumulative \
qtimlqnn name=stage_03_inference model=/etc/models/
face_attrib_net_quantized.bin backend=/usr/lib/libQnnHtp.so \
qtiqmmfsrc video_0::type=video name=camsrc ! video/x-
raw,format=NV12,width=1920,height=1080 ! queue ! waylandsink
fullscreen=true sync=false \
camsrc.image_1 ! video/x-raw,width=1920,height=1080 ! qtivtransform !
video/x-raw,format=NV12 ! tee name=t_split_1 \
t_split_1. ! queue ! metamux_1. \
t_split_1. ! queue ! stage_01_preproc. stage_01_preproc. ! queue !
stage_01_inference. stage_01_inference. ! queue ! stage_01_postproc.
stage_01_postproc. ! text/x-raw ! queue ! metamux_1. \
qtimetamux name=metamux_1 ! queue ! tee name=t_split_3 \
t_split_3. ! queue ! stage_03_preproc. stage_03_preproc. ! queue !
stage_03_inference. stage_03_inference. ! queue ! multifilesink
location=/etc/data/tensor_%d.bin sync=true async=false enable-last-
sample=false
```

   A list of options is displayed.

2. To prepare for capturing a facial image, do the following:
   a. Select the following options from the list:
      i. `(3)PLAYING`: Move the pipeline to the Playing state.
      ii. `(p)Plugin Mode` ➔ `(26)camsrc`➔ `(37)capture-image`: Capture the image using a camera source.
   b. Using the live preview on the display, face the camera and ensure that the camera is pointed straight and there is only one person in the frame.
   c. In the terminal, enter 1 for the following values:
      i. `GstImageCaptureMode` for `arg0`.
      ii. `guint` for `arg1`.
3. To capture all the sides of your face, select `capture-image` do the following for each side:
   a. Left and right: Turn your head left by 40° while keeping the landmarks visible, then repeat steps 3 and 4. Turn your head right (by 40°) and repeat.
   b. Up and down: Raise your head by 30° while keeping the landmarks visible, then repeat steps 3 and 4. Lower your head (by 30°) and repeat.
4. To stop the pipeline, use `(b)Back` and `(q)Quit`.

   After running the pipeline, five individual tensor bins are created (`tensor_0.bin` to `tensor_4.bin`) with facial properties recorded for each side of the face.
5. On the target device, go to`/etc/data/`, find the tensor bins. To pull the bins from the target device to the Linux host computer, run the following commands:

```
scp root@<IP-Address>:/etc/data/tensor_0.bin .
scp root@<IP-Address>:/etc/data/tensor_1.bin .
scp root@<IP-Address>:/etc/data/tensor_2.bin .
scp root@<IP-Address>:/etc/data/tensor_3.bin .
scp root@<IP-Address>:/etc/data/tensor_4.bin .
```

6. To merge the tensor bins with all the facial properties into a cohesive image, download and run the `facedb.py` script in the same directory as the tensor bins on the Linux host computer.
   a. Download the `facedb.py` script:

```
curl -L -O https://raw.githubusercontent.com/quic/sample-apps-for-qualcomm-linux/refs/heads/main/scripts/facedb.py
```

   b. Run the script. Note that *`<Name of the person>`* is case and style sensitive. Ensure that you use the same name consistently.

```
python3 ./facedb.py "<Name of the person>" 512 32 tensor_0.bin tensor_1.bin tensor_2.bin tensor_3.bin tensor_4.bin
```

   A `face.bin` binary is created.
7. Push the `face.bin` binary to`/etc/data` directory and rename it to `face0.bin`.

```
scp face.bin root@<IP address of target device>:/etc/data/face0.bin
```

8. To generate the `face_recognition.labels` file and register the new person into the database, use the following reference label file for two-person registered face:

```
(structure)"<Name of the person>,id=(guint)0x0,database=(string)/etc/data/
face0.bin,color=(guint)0x00FF00FF;"
```

```
(structure)"<Name of the person>,id=(guint)0x01,database=(string)/etc/data/
face1.bin,color=(guint)0x00FF00FF;"
```

NOTE Update the ID field according to the number in the list. If more faces are registered, add the structure in a new line within `face_recognition.labels`.

9. To save the labels, push the updated `face_recognition.labels` file to the `/etc/labels` directory on the target device.

```
scp face_recognition.labels root@<IP address of target device>:/etc/labels
```

Run the application

NOTE The following commands provide the default model and label paths. If you have a different folder structure, replace the default paths in the command-line parameters. See Sample model and label files.

The sample application uses the `/etc/configs/config-face-recognition.json` file to read the input parameters.

To create your own config JSON file, use config-face-recognition.json as a reference.

1. Use the following format of the `/etc/configs/config-face-recognition.json` file:

```
{
"ml-framework": "<qnn or tflite>",
"face-detection-model": "<path for face detection model file>",
"face-landmark-model":"<path for face landmark model file>",
"face-recognition-model":"<path for face recognition model file>",
"face-detection-labels":"<path for face detection labels file>",
"face-landmark-labels":"<path for face landmark labels file>",
"face-recognition-labels":"<path for face recognition labels file>",
"face-detection-constants":"<Model-constants-for-face-detection-litert
models>",
"face-landmark-constants":"<Model-constants-for-face-landmark-litert-
model>",
"face-recognition-constants":"<Model-constants for-face-recognition-litert-
model>"
}
```

NOTE Update the config JSON file based on the model, input stream, and other properties. For more information, see Config JSON field description.

For example, run the application using LiteRT, custom models, custom labels file:

```
{
  "ml-framework":"tflite",
  "face-detection-model":"/etc/models/face_det_lite_quantized.tflite",
  "face-landmark-model":"/etc/models/facemap_3dmm_quantized.tflite",
```

```
  "face-recognition-model":"/etc/models/face_attrib_net_quantized.tflite",
  "face-detection-labels": "/etc/labels/face_detection.labels",
  "face-landmark-labels": "/etc/labels/face_landmark.labels",
  "face-recognition-labels": "/etc/labels/face_recognition.labels",
  "face-detection-constants": "DET,q-offsets=<178.0, 0.0, 102.0>,q-
scales=<0.034
  "face-landmark-constants":"DMM,q-offsets=<211.0>,q-
scales=<0.06002333015203476
  "face-recognition-constants":"qfr,q-offsets=<124.0, 153.0, 125.0, 133.0,
126.0
}
```

2. Run the gst-ai-face-recognition application:

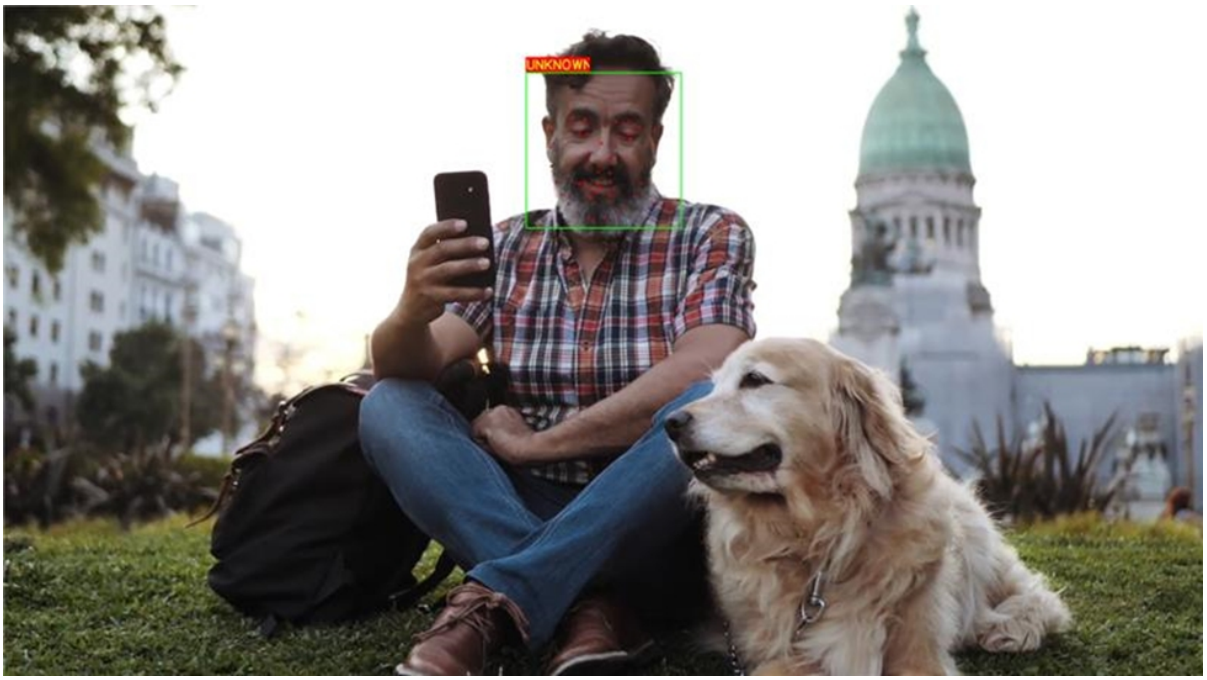
```
gst-ai-face-recognition --config-file=/etc/configs/config-face-
recognition.json
```

To display the available help options, run the following commands in the SSH shell:

```
gst-ai-face-recognition -h
```

To stop the use case, use **CTRL + C**.

Expected output



Pipeline flow

The table lists the plugins used in the face recognition pipeline:

Plugin	Description
Camera source:qtiqmmfsrc	■ Captures the live stream from camera. ■ Uses tee to split the stream for inferencing.
File source: filesrc	■ Captures the video stream using filesrc, followed by qtdemux, which demultiplexes the stream. ■ Uses tee to split the stream for inferencing.
RTSP source: rtspsrc	■ Captures the RTSP stream using rtspsrc, followed by rtph264depay for video extraction. ■ Uses tee to split the stream for inferencing.
h264parse	Parses the H.264 video.
v4l2h264dec	Decodes the video.
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs the following preprocessing on the stream data. This preprocessing is done when the model expects floating-point values as input. a. Color conversion b. Scaling (up or down) c. Normalization 3. Converts the preprocessed video stream to a tensor stream on its source pad. The tensor stream is used for inferencing in the later stages of the pipeline.
qtimlqnn	Acts as the inferencing plugin for Qualcomm Neural Network model. 1. After the inference runtime receives the tensor stream on its sink pad, it runs the inference. 2. Produces a tensor stream with the inference results on its source pad.
qtimlvdetection	■ Handles inference results from any face detection model. ■ Applies a threshold to the chosen number of results.
qtimetamux	■ Receives the output of the face detection models from qtimlvdetection and multiplexes it. ■ Receives the output of facial pose from qtimlvpose and multiplexes it.
tee	Splits the stream for inferencing.
qtimlvpose	Performs the facial pose recognition.
qtimlvclassification	Receives the stream from qtimetamux and classifies the face.
qtivoverlay	1. Receives the multiplexed stream. 2. Overlays the bounding boxes on the stream.
Waylandsink	1. Waylandsink submits the video stream received on its sink pad to Weston. 2. Weston renders the video stream on a local display.

Config JSON field description

Table 2-24 Field description–config-face-recognition.json file

Field	Values/description
ml-framework	Use one of the following models:

Table 2-24 Field description–config-face-recognition.json file (cont.)

Field	Values/description
	▪ `tflite`–LiteRT ▪ `qnn`–Qualcomm AI Engine direct
Models and labels	See Sample model and label files. ▪ **face-detection-model**: The path to the face detection model. ▪ **face-landmark-model**: The path to the face landmark model. ▪ **face-recognition-model**: The path to the face recognition model. ▪ **face-detection-labels**: The path to the face detection labels. ▪ **face-landmark-labels**: The path to the face landmark labels. ▪ **face-recognition-labels**: The path to the face recognition labels.
Constants	▪ **face-detection-constants**: The face detection model constants. ▪ **face-landmark-constants**: The face landmark model constants. ▪ **face-recognition-constants** The face recognition model constants.

Known issues

The 3D facial landmark isn't observed with LiteRT models.

RELATED INFORMATION

"Object detection" on page 33
"Pose detection" on page 38

2.1.18 Audio classification

The **gst-ai-audio-classification** application shows the audio classification using the input from a file source and microphone. The classification results and video preview are displayed.

The figure shows the pipeline, which gets the input from a file or a microphone, does the preprocessing and inferencing on the AI hardware. The results are displayed on the screen.

For information about the plugins used in the pipeline flow, see Pipeline flow.

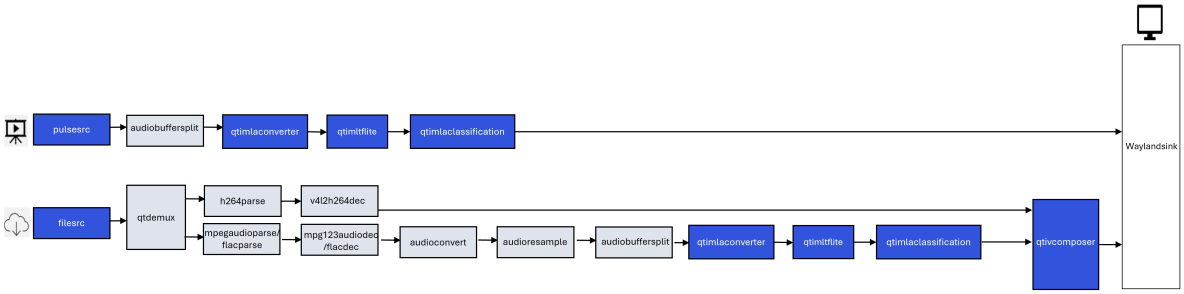


Figure 2-31 gst-ai-audio-classification pipeline

Sample model and label files

Table 2-25 Sample model and label files for gst-ai-audio-classification

Runtime	Model files	Label files
LiteRT	*yamnet.tflite*	*yamnet.labels*

Prerequisites

- If not already done so, Download and install eSDK.
- Download model and label files.
- To access your host computer, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is already enabled, you can skip this step.
- Push the model files from the Linux host.

  ```
  scp <model_filename> root@<IP address of target device>:/etc/models
  ```
- Note that the Download model and label files downloads the sample `video.mp4` video to the `/etc/media` directory. If you are using a custom video, then ensure that you push the video to `/etc/media` and update the file path in the application config. JSON file.
- Ensure that the video in use has an audio component along with the video.
- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.
- Enable the display:

  ```
  export XDG_RUNTIME_DIR=/dev/socket/weston && export
  WAYLAND_DISPLAY=wayland-1
  ```

Run the application

The sample application uses the `/etc/configs/config-audio-classification.json` file to read the input parameters.

To create your own config JSON file, use config-audio-classification.json as a reference.

1. Use the following format of the `config-audio-classification.json` file.

   ```
   {
     "file-path": "<path to video file>",
     "model": "<path to model file>",
     "labels": "<path to label file>",
     "threshold": <integer between 1 and 100>,
     "runtime": "<cpu or gpu>",
     "codec": "<mp3 or flac>"
   }
   ```

   NOTE Update the config JSON file based on the model, input stream, and other properties. For more information, see Config JSON field description.

For example, run the application using input from a file, MP3 encoding, and GPU runtime:

```
{
"file-path": "/etc/media/video-mp3.mp4",
"model": "/etc/models/yamnet.tflite",
"labels": "/etc/labels/yamnet.labels",
"runtime": "gpu",
"threshold": 20,
"codec": "mp3"
}
```

2. Run the gst-ai-audio-classification application:

```
gst-ai-audio-classification --config-file=/etc/configs/config-audio-classification.json
```

To display the available help options, run the following command in the SSH shell:

```
gst-ai-audio-classification -h
```

To stop the use case, use **CTRL + C**.

Expected output

The output video and classified audio are played on the screen.

Pipeline flow

The table lists the plugins used in the audio classification pipeline:

Plugin	Description
File source: filesrc	▪ Captures the video stream using filesrc, followed by qtdemux, which demultiplexes the stream. ▪ Uses tee to split the stream for inferencing.
h264parse	Parses the H.264 video.
v4l2h264dec	Decodes the video.
mpegaudioparse or flacparse	Parses the audio (MP3 or FLAC).
mpg123audiodec or flacdec	Decodes the audio (MP3 or FLAC).
audioconvert	Converts the audio buffers between various possible formats.
audioresample	Resamples the audio buffers to different sample rates.
pulsesrc	Reads the audio from the microphone.
audiobuffersplit	Splits the incoming audio buffers into equal sized chunks.
qtimlaconverter	Performs preprocessing on the audio stream and converts the stream to a tensor stream. The audio classification model uses this tensor stream for inferencing.
qtimltflite	Performs inferencing using the YAMNet model.
qtimlaclassification	Handles the audio classification inference results: 1. Applies a threshold to the chosen number of results. 2. Creates text overlay for classes.

Plugin	Description
qtivcomposer	Combines the text overlay for classification results and video preview.
Waylandsink	1. Waylandsink submits the video stream received on its sink pad to Weston. 2. Weston renders the video stream on a local display.

Config JSON field description

Field	Values/description
runtime	Use one of the following runtimes: ▪ `cpu` ▪ `gpu`
Input source	Use one of the following input sources: ▪ `file-path`: The directory path to the video file. ▪ Microphone
threshold=<integer>	Use any integer between 1 and 100.
codec	The audio codec of input video: ▪ MP3 (default) ▪ FLAC

RELATED INFORMATION

"Audio classification decode and display with LiteRT" on page 221

2.1.19 Metadata parsing

The **gst-ai-metadata-parser-example** application takes the live video stream input from camera, file, or RTSP source, and passes the stream to the YOLO models for object detection and preview. The overlaid AI model output with label and bounding boxes is displayed on an HDMI display. The extracted metadata is logged in the console and used to count the number of humans in the frame.

The figure shows the pipeline for metadata parsing. For information about the plugins used in the pipeline flow, see Pipeline flow.

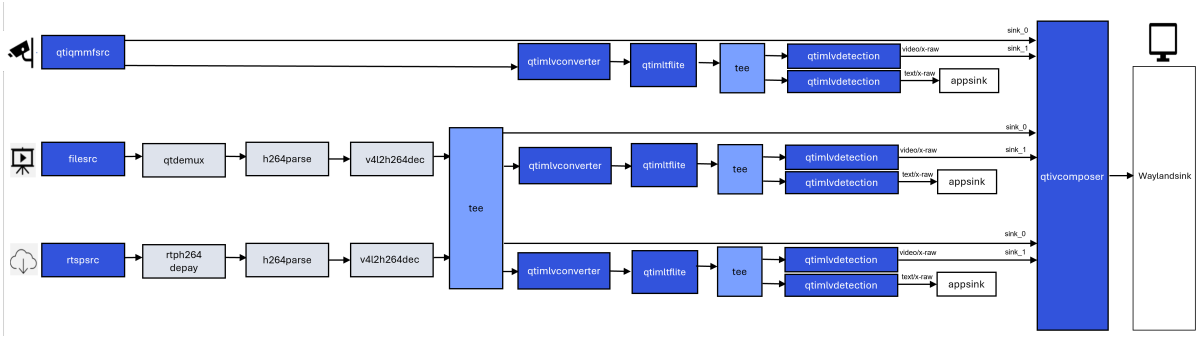


Figure 2-32 gst-ai-metadata-parser-example pipeline

Sample model and label files

Table 2-26 Sample model and label files for gst-ai-object-detection

Runtime	Model files	Label files
LiteRT	*yolov8_det_quantized.tflite*	*yolov8.labels*

Prerequisites

- If not already done so, Download and install eSDK.
- Download model and label files.
- To access your host computer, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is already enabled, you can skip this step.
- Push the model files from the Linux host computer.

  ```
  scp <model_filename> root@<IP address of target device>:/etc/models
  ```
- Note that the Download model and label files downloads the sample `video.mp4` video to the `/etc/media` directory. If you are using a custom video, then ensure that you push the video to `/etc/media` and update the file path in the application config.JSON file.
- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.
- Enable the display:

  ```
  export XDG_RUNTIME_DIR=/dev/socket/weston && export
  WAYLAND_DISPLAY=wayland-1
  ```

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

NOTE To view the bounding box information along with the human count, run the following command before running the application:

```
export GST_DEBUG=4
```

Run the application

Use the following command to run the application:

```
gst-ai-metadata-parser-example --config-file=/etc/configs/config-metadata-
parser.json
```

The sample application uses the `/etc/configs/config-metadata-parser.json` file to read the input parameters.

To create your own config JSON file, use config-metadata-parser.json as a reference.

1. Use the following format of the `config-metadata-parser.json` file.

   ```
   {
     "file-path": "<path to video file>",
     "model": "<path to model file>",
     "labels": "<path to label file>",
     "constants" : "<model constants for LiteRT model>",
   ```

```
  "threshold": <integer between 1 and 100>,
  "runtime": "<cpu, gpu, or dsp runtime>"
}
```

NOTE Update the config JSON file based on the model, input stream, and other properties. For more information, see Config JSON field description.

For example, run the application using input from a camera, YOLOv8 model, and human count on console:

```
{
"file-path": "/etc/media/video.mp4",
"model": "/etc/models/yolov8_det_quantized.tflite",
"labels": "/etc/labels/yolov8.labels",
"threshold": 40,
"runtime": "dsp"
}
```

2. Run the gst-ai-metadata-parser-example application:

```
gst-ai-metadata-parser-example --config-file=/etc/configs/config-metadata-parser.json
```

To display the available help options, run the following command in the SSH shell:

```
gst-ai-metadata-parser-example -h
```

To stop the use case, use **CTRL + C**.

Expected output

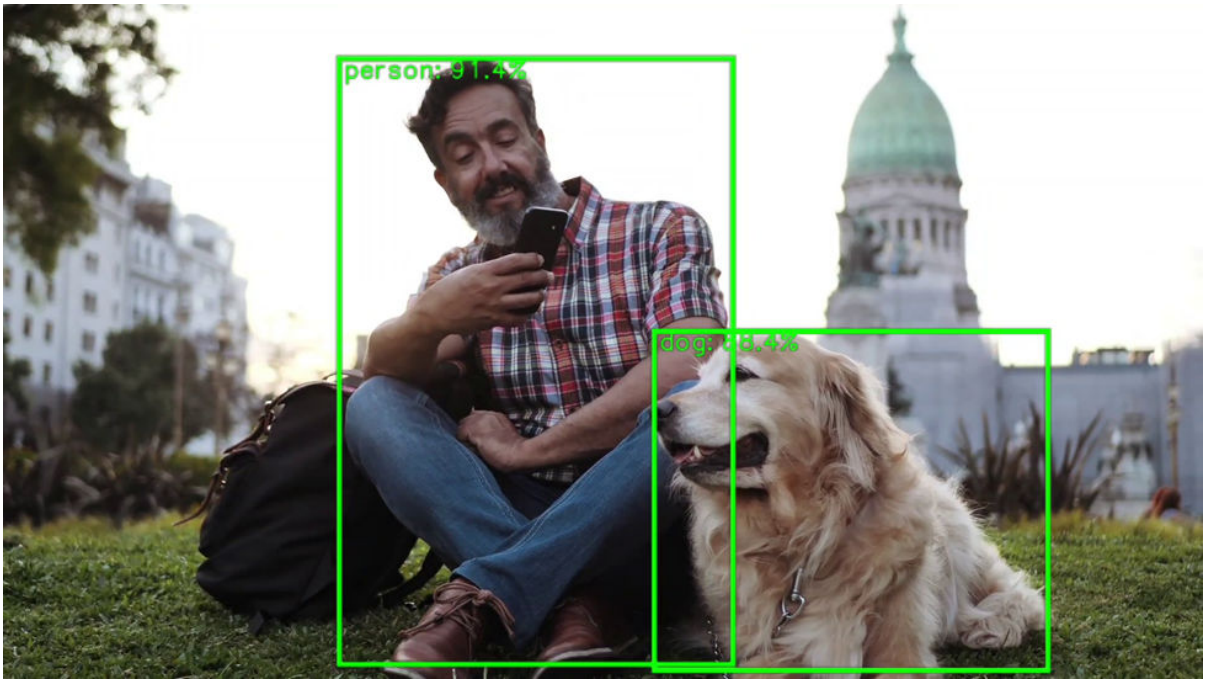


Figure 2-33 Expected output for gst-ai-metadata-parser-example application–object detection

Pipeline flow

The table lists the plugins used in the metadata parsing pipeline:

Plugin	Description
Camera source:qtiqmmfsrc	▪ Captures the live stream from camera. ▪ Uses tee to split the stream into two for inferencing and composing.
File source: filesrc	▪ Captures the video stream using filesrc, followed by qtdemux, which demultiplexes the stream. ▪ Uses tee to split the stream into two for inferencing and composing.
RTSP source: rtspsrc	▪ Captures the RTSP stream using rtspsrc, followed by rtph264depay for video extraction. ▪ Uses tee to split the stream into two for inferencing and composing.
h264parse	Parses the H.264 video.
v4l2h264dec	Decodes the video.
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs the following preprocessing on the stream data. This preprocessing is done when the model expects floating-point values as input. a. Color conversion b. Scaling (up or down) c. Normalization 3. Converts the preprocessed video stream to a tensor stream on its source pad. The tensor stream is used for inferencing in the later stages of the pipeline.
Inferencing plugin:qtimltflite	1. After the inference runtime receives the tensor stream on its sink pad, it runs the inference. 2. Produces a tensor stream with the inference results on its source pad.
qtimlvdetection	The first qtimlvdetection plugin handles inference results from any object detection model. 1. Applies a threshold to the chosen number of results. 2. Loads the YOLOv8 module. 3. Produces video frames with only bounding boxes that can be overlaid on objects. 4. Sends these processed frames to the sink pad of qtivcomposer.
	The second qtimlvdetection plugin does the following: ▪ Produces the output in a text format (bounding box coordinates and labels). ▪ This output is connected to appsink plugin where the metadata is read, parsed, and logged. ▪ The bounding box information is used to count number of humans in each frame.
qtivcomposer	1. Composes frames with contents from its sink pads. 2. Pushes the GStreamer buffers containing these composed frames to its source pad.
Waylandsink	1. Waylandsink submits the video stream received on its sink pad to Weston. 2. Weston renders the video stream on a local display.

Config JSON field description

Table 2-27 Field description–gst-ai-metadata-parser-example file

Field	Values/description
runtime	Use one of the following runtimes: ▪ `cpu` ▪ `gpu` ▪ `dsp`
Input source	Use one of the following input sources: ▪ `camera`–Primary (0) or secondary (1). ▪ `file-path`–The directory path to the video file. ▪ `rtsp-ip-port`–The address of the RTSP stream: *`rtsp://<ip>:<port>/<stream>`*
constants	Add the constants, offsets, and coefficients used by the chosen module. `"module,q-offsets=<value>,q-scales=<value>;"`

RELATED INFORMATION

"Object detection" on page 33

2.1.20 AI USB camera

The **gst-ai-usb-camera-app** streams the video from a USB webcam connected to the Qualcomm EVK. This webcam should be accessible as a `/dev/videoX` device. Additionally, you can perform object detection and preview the results.

You can choose to preview the output, save the video encoder output, save the camera YUV to a file, live stream through RTSP, or perform object detection and preview the results.

NOTE The video encoding use case isn't supported in the current release.

The figures show pipeline, which processes the input from the USB camera to generate various outputs.

For more information about the plugins used in these pipelines, see Pipeline flow.

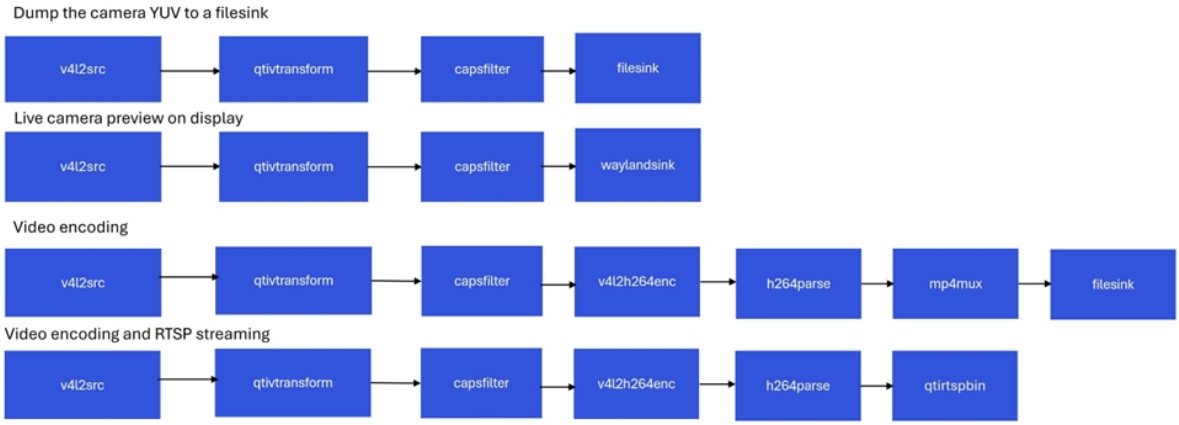


Figure 2-34 gst-ai-usb-single-camera-app pipeline

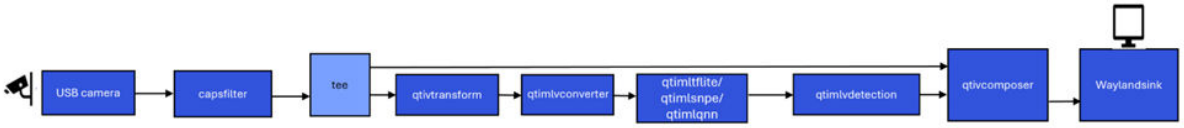


Figure 2-35 gst-ai-usb-single-camera-app pipeline with object detection

Sample model and label files

Table 2-28 Sample model and label files for gst-ai-usb-camera-app

Runtime	Model files	Label files
Qualcomm Neural Processing SDK	*yolonas.dlc*	▪ *yolonas.labels* ▪ *yolov8.labels*
LiteRT	▪ *yolov8_det_quantized.tflite* ▪ *yolonas_quantized.tflite*	
Qualcomm AI Engine direct	*yolov8_det_quantized.bin*	

Prerequisites

- If not already done so, Download and install eSDK.
- To run the application, push the model and label files to the device. For instructions, see Download model and label files.

  The application supports the Qualcomm Neural Processing SDK, Qualcomm AI Engine direct, and LiteRT models.
- To access your host computer, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is already enabled, you can skip this step.
- Push the model files from the Linux host computer:

  ```
  scp <model_filename> root@<IP address of target device>:/etc/models
  ```
- Note that the Download model and label files downloads the sample `video.mp4` video to the `/etc/media` directory. If you are using a custom video, then ensure that you push the video to `/etc/media` and update the file path in the application config.JSON file.
- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.
- Enable the display:

  ```
  export XDG_RUNTIME_DIR=/dev/socket/weston && export
  WAYLAND_DISPLAY=wayland-1
  ```

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

- Ensure that the USB webcam is connected to the device.

  NOTE If the USB camera isn't detected on the device, Download the required firmware. See Download PCIe to USB controller firmware.

Run the application

The sample application uses the `/etc/configs/config-usb-camera-app.json` file to read the input parameters. Ensure that you use the config JSON file to run the application.

To create your own config JSON file, use config-usb-camera-app.json as a reference.

1. Enter the SSH shell and copy the YOLO-NAS label files to YOLOv8:

   ```
   cp /etc/labels/yolonas.labels /etc/labels/yolov8.labels
   ```

2. Use the following format of the `/etc/configs/config-usb-camera-app.json` file:

   ```
   {
     "width": "<Supported USB camera width>",
     "height": "<Supported USB camera Height>",
     "framerate": "<Supported USB camera fps>",
     "format": "<Supported USB camera width>",
     "output": "<USB camera output type>",
     "ip-address":"<Device IP address in case of rtsp streaming>",
     "port":"<Device port num in case of rtsp streaming>",
     "enable-object-detection": "<to enable the object-detection>",
     "file-path": "<input video path>",
     "ml-framework": "<snpe, or tflite or qnn framework>",
     "yolo-model-type": "<yolov8 or yolonas or yolov5>",
     "model": "<Model Path>",
     "labels": "<Label Path>",
     "constants": "<Model Constants for LiteRT Model>",
     "threshold": <Post-processing threshold, integer value from 1-100>,
     "runtime": "<dsp, cpu or gpu runtime>"
    }
   ```

   NOTE Update the config JSON file based on the model, input stream, and other properties. For more information, see Config JSON field description.

   For example, run the application using RTSP input, LiteRT, YOLOv8 model, DSP runtime, custom constants, and custom threshold value.

   In the `config-usb-camera-app.json` file, you can replace `model-detection.tflite` with `yolov8_det_quantized.tflite` and `detection.labels` with `yolov8.labels`.

   ```
    {
     "width": 640,
     "height": 480,
     "framerate": 30,
     "format": "YUY2",
     "output": "PREVIEW",
     "ip-address":"127.0.0.1",
     "port":"8900",
     "enable-object-detection": "TRUE",
     "ml-framework": "tflite",
     "yolo-model-type": "yolov8",
     "model": "/etc/models/model-detection.tflite",
   ```

```
  "labels": "/etc/labels/detection.labels",
  "constants": "YOLOv8,q-offsets=<21.0, 0.0, 0.0>,q-
scales=<3.093529462814331, 0.00390625, 1.0>;",
  "threshold": 40,
  "runtime": "dsp"
}
```

3. Run the gst-ai-usb-camera-app application:

```
gst-ai-usb-camera-app --od-config-file=/etc/configs/config-usb-camera-
app.json
```

NOTE You can run the `Yolo-NAS-Quantized.tflite` model using the YOLOv8 value in the `yolo-model-type` field.

To display the available help options, run the following command in the SSH shell:

```
gst-ai-usb-camera-app -h
```

To stop the use case, use **CTRL + C**.

Expected output

The table lists the expected output for each use case:

Table 2-29 Expected output for gst-usb-single-camera-app

Use case	Output
Preview	Preview of the camera stream. See Figure 2-36.
Save the MP4 and YUV data	Save the output to a file at `/etc/` folder path.
RTSP	View the output on a remote display.
Object detection	See Figure 2-37

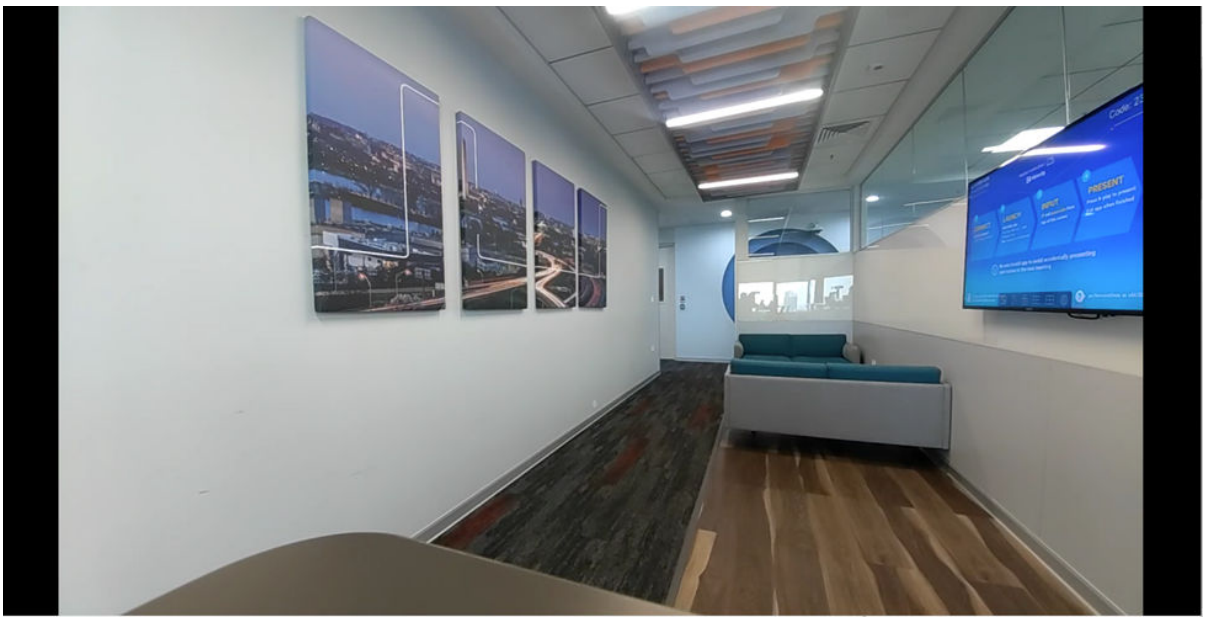


Figure 2-36 Expected output for gst-ai-usb-camera-app application–Preview

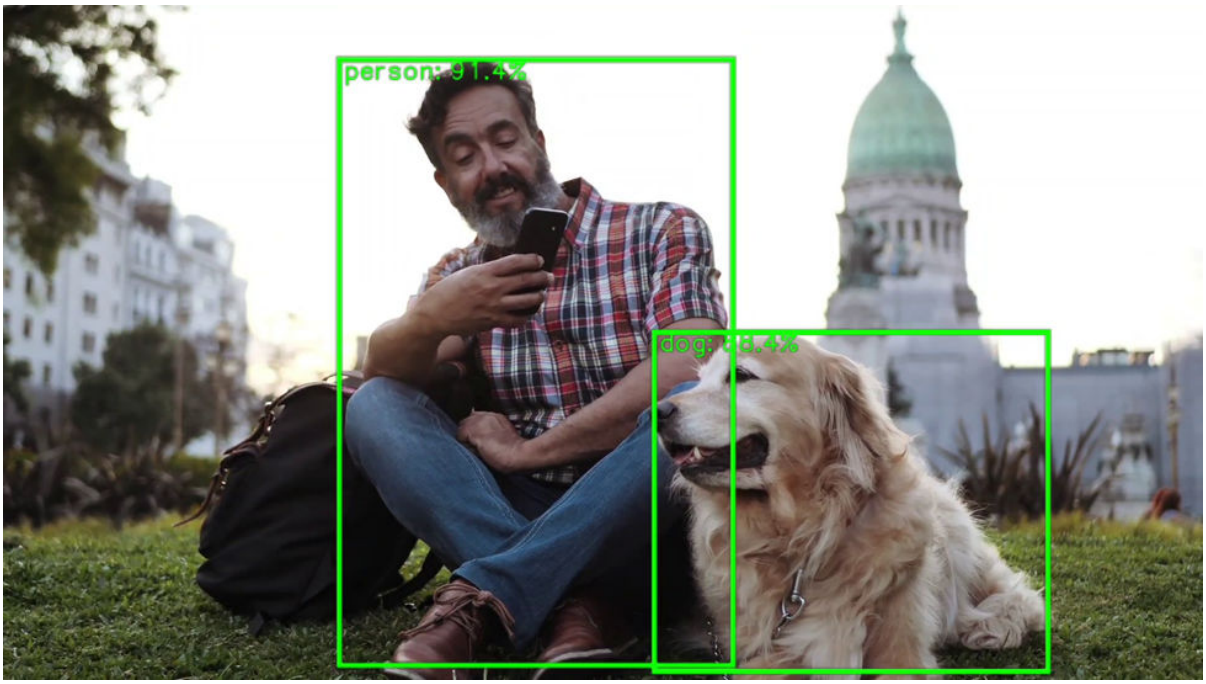


Figure 2-37 Expected output for gst-ai-usb-camera-app application–object detection

Pipeline flow

The table lists the plugins used in AI USB camera pipelines:

Pipeline	Description
Dump the camera YUV to a filesink	1. USB camera captures the camera live stream. 2. qtivtransform transforms the stream data.

Pipeline	Description
	3. Capsfilter is applied to enforce constraints on the raw video data. 4. Filesink is used to dump the YUV data into a file.
Preview on display	1. USB camera captures the camera live stream. 2. qtivtransform transforms the stream data. 3. Capsfilter is applied to enforce constraints on the raw video data. 4. The data is sent to the Wayland display sink (Waylandsink) for live preview.
Video encoding	1. USB camera captures the camera live stream. 2. qtivtransform transforms the stream data. 3. Capsfilter is applied to enforce constraints on the raw video data. 4. v4l2h264enc is used to encode the video using the H.264 format. 5. H264parse is used to parse the video. 6. Mp4mux is used to multiplex the video into an MP4 container. 7. Filesink is used to write the video to a file.
Video encoding and RTSP streaming	1. USB camera captures the camera live stream. 2. qtivtransform transforms the stream data. 3. Capsfilter is applied to enforce constraints on the raw video data. 4. v4l2h264enc is used to encode the video using the H.264 format. 5. H264parse is used to parse the video. 6. qtirtspbin is used to load the stream to RTSP.
Camera capture and object detection	1. USB camera captures the camera live stream. 2. Capsfilter is applied to enforce constraints on the raw video data. 3. tee is used to split the stream for inferencing. 4. qtivtransform transforms the stream data. 5. qtimlvconverter does preprocessing and converts the video stream to a tensor stream, which is used for inferencing. 6. qtimlsnpe, qtimltflite, or qtimlqnn run the inference on the stream. 7. qtimlvdetection handles the inference results from any object detection model and produces video frames. 8. qtivcomposer composes the video frames and shares them with Waylandsink. 9. Waylandsink submits the composed video stream to Weston, which renders it on the local display.

Config JSON field description

Table 2-30 Field description–config_detection.json file

Field	Values/description
ml-framework	Use one of the following models: ▪ `snpe`–Qualcomm Neural Processing SDK ▪ `tflite`–LiteRT ▪ `qnn`–Qualcomm AI Engine direct
yolo-model-type	Run the `yolov5`, `yolov8`, and `yolonas` models respectively. See Sample model and label files.
runtime	Use one of the following runtimes:

Table 2-30 Field description–config_detection.json file (cont.)

Field	Values/description
	▪ `cpu` ▪ `gpu` ▪ `dsp`
Input source	Use one of the following input sources: ▪ `camera`–Primary (0) or secondary (1). ▪ `file-path`–The directory path to the video file. ▪ `rtsp-ip-port`–The address of the RTSP stream: *`rtsp://<ip>:<port>/<stream>`*

Known issues

The USB camera only supports 640 × 480 resolution.

RELATED INFORMATION

"Object detection" on page 33

2.2 Run multimedia sample applications

The multimedia sample applications provide custom use cases that realize the camera, video, and audio capabilities of Qualcomm Linux.

NOTE For camera and video sample applications that require a YUV player, you can use https://sourceforge.net/projects/raw-yuvplayer/.

Troubleshooting

- **Camera**: For any application, if the camera service doesn't work or fails during execution, see Camera troubleshooting to restart it.
- **Display**: If there are issues enabling the display, do the following:
  a. Open two SSH shells–one to enable Wayland and the other to run the use cases.
  ```
  ssh root@<ip-addr of the target device>
  ```
  b. Enable Weston in one SSH shell and keep it running:
  ```
  export GBM_BACKEND=msm && export XDG_RUNTIME_DIR=/dev/socket/weston &&
  mkdir -p $XDG_RUNTIME_DIR && weston --continue-without-input --idle-
  time=0
  ```
  c. Enable display in the other SSH shell:
  ```
  export XDG_RUNTIME_DIR=/dev/socket/weston && export
  WAYLAND_DISPLAY=wayland-1
  ```

See also

- Camera Guide
- Video Guide
- Audio Guide

2.2.1 Run sample applications for camera

The camera sample applications provide custom use cases for single stream and multistream scenarios, previews and snapshots, dynamic updates to configurations and properties, and various other capabilities of the Qualcomm Linux camera.

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

2.2.1.1 Single camera streaming

The **gst-camera-single-stream-example** application allows you to use a single camera with various outputs, including preview, encode, YUV dump, and RTSP streaming.

For example, you can use this application in a security system where the goal is to capture a video from one camera and stream it across a network. You can either view it or archive it for future review.

The figure shows the pipeline flow for various single camera stream use cases. For more information, see Pipeline flow.

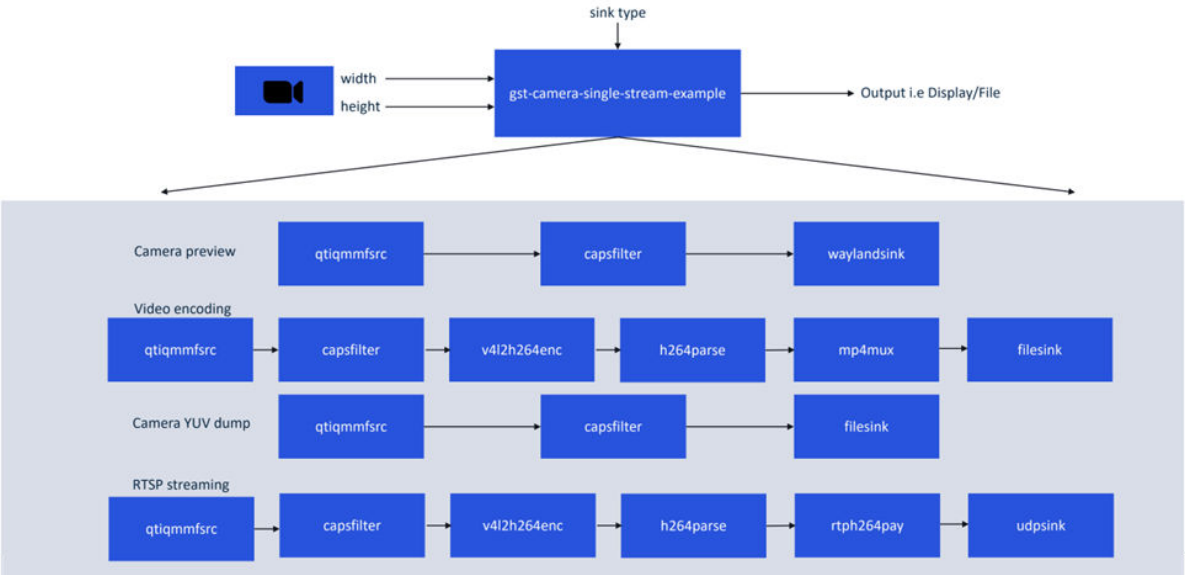


Figure 2-38 gst-camera-single-stream-example pipeline

Prerequisites

- If not already done so, Download and install eSDK.
- To access your host device, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is enabled, you can skip this step.
- Push the files from the Linux host computer.

  ```
  scp <filename> root@<IP address of target device>:/etc/
  ```
- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.

- Enable the display:

  ```
  export XDG_RUNTIME_DIR=/dev/socket/weston && export
  WAYLAND_DISPLAY=wayland-1
  ```

- Connect to the speaker to enable the audio. For instructions, see Set up audio hardware.

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

Run the application

- View the live camera on Waylandsink:

  ```
  gst-camera-single-stream-example -o 0 -w 1920 -h 1080
  ```

- Dump the MP4 to the `/opt/` folder:

  ```
  gst-camera-single-stream-example -o 1 -w 1920 -h 1080
  ```

- Dump the YUV data to the `/opt/` folder:

  ```
  gst-camera-single-stream-example -o 2 -w 1920 -h 1080
  ```

- View the RTSP stream (run the RTSP server) on the Linux host:
  a. Run the RTSP server in a separate console on target with `udpsrc` (you can run it in the background as a service):

     ```
     gst-rtsp-server -p 8900 -a <ip_address> -m /live "( udpsrc name=pay0
     port=<port_num> caps=\"application/x-rtp,media=video,clock-
     rate=90000,encoding-name=H264,payload=96\" )"
     ```

     For example:

     ```
     gst-rtsp-server -p 8900 -a 10.92.182.47 -m /live "( udpsrc name=pay0
     port=8554 caps=\"application/x-rtp,media=video,clock-
     rate=90000,encoding-name=H264,payload=96\" )"
     ```

  b. Run the use case:

     ```
     gst-camera-single-stream-example -o 3 -w 1280 -h 720 -i <ip_address> -p
     <port_num>
     ```

To view the stream on the VLC player or FFplay, run the respective command on the Linux host:

- **For VLC:**

  ```
  vlc -vvv rtsp://<ip_address>:8900/live
  ```

  For FFplay:

  ```
  ffplay rtsp://<ip_address>:8900/live
  ```

To display the available help options, run the following command in the SSH shell:

```
gst-camera-single-stream-example --help
```

To stop the use case, use **CTRL + C**.

Expected output

Based on the selected output type, the following table lists the expected output:

Table 2-31 Expected output for gst-camera-single-stream-example

Output	Description
Display	Preview the stream.
Dump the MP4 and YUV data	Save the output to a file at `/opt/`.
RTSP	View the output on a remote display.

Use the following command to pull the files from host computer:

```
scp root@<IP address of target device>:/opt/<filename to pull> <destination
directory>
```

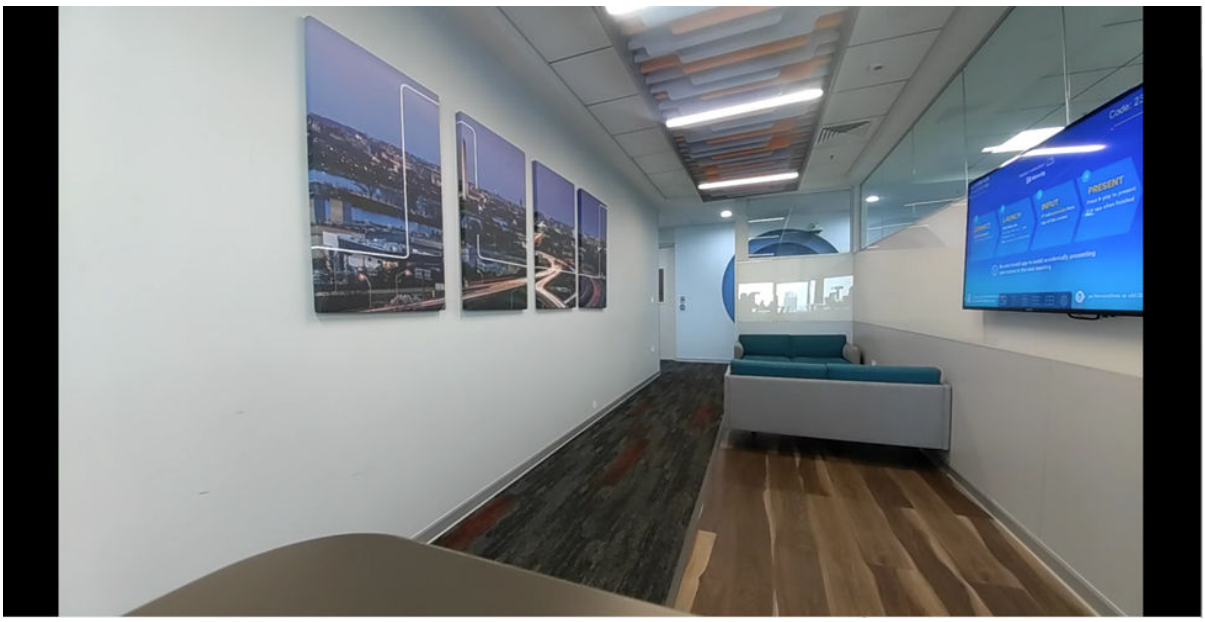


Figure 2-39 Expected output for gst-camera-single-stream-example application

Pipeline flow

The table lists the plugins used in the single camera streaming pipeline:

Pipeline	Description
Live camera preview on display	1. qtiqmmfsrc captures the camera live stream. 2. Capsfilter is applied to enforce constraints on the raw video data. 3. The data is sent to the Wayland display sink (Waylandsink) for live preview.
Video encoding	1. qtiqmmfsrc captures the camera live stream. 2. Capsfilter is applied to enforce constraints on the raw video data. 3. The data is sent to the Wayland display sink (Waylandsink) for live preview. 4. v4l2h264enc is used to encode the video using the H.264 format. 5. H264parse is used to parse the video.

Pipeline	Description
	6. Mp4mux is used to multiplex the video into an MP4 container. 7. Filesink is used to write the video to a file.
Dump the camera YUV to a filesink	1. qtiqmmfsrc captures the camera live stream. 2. Capsfilter is applied to enforce constraints on the raw video data. 3. Filesink is used to dump the YUV data into a file.
RTSP streaming	1. qtiqmmfsrc captures the camera live stream. 2. Capsfilter is applied to enforce constraints on the raw video data. 3. v4l2h264enc is used to encode the video using the H.264 format. 4. H264parse is used to parse the video, rtph264pay is used to load the stream to RTSP, and then the stream is sent to a UDP sink for RTSP streaming.

RELATED INFORMATION

"Camera streaming using OpenCV" on page 176

2.2.1.2 USB camera

The **gst-ai-usb-camera-app** streams the video from a USB webcam connected to the Qualcomm EVK. This webcam should be accessible as a `/dev/videoX` device. While you can perform object detection and preview the results, this procedure allows you to process the input from the USB camera without the AI inferencing.

The AI USB camera application uses the `/etc/config/config-usb-camera-app.json` file to read the input parameters.

You can choose to run this application with and without the AI inferencing. To run this application with AI inferencing, see AI USB camera.

The figures show pipeline, which processes the input from the USB camera to generate various outputs.

For more information about the plugins used in these pipelines, see Pipeline flow.

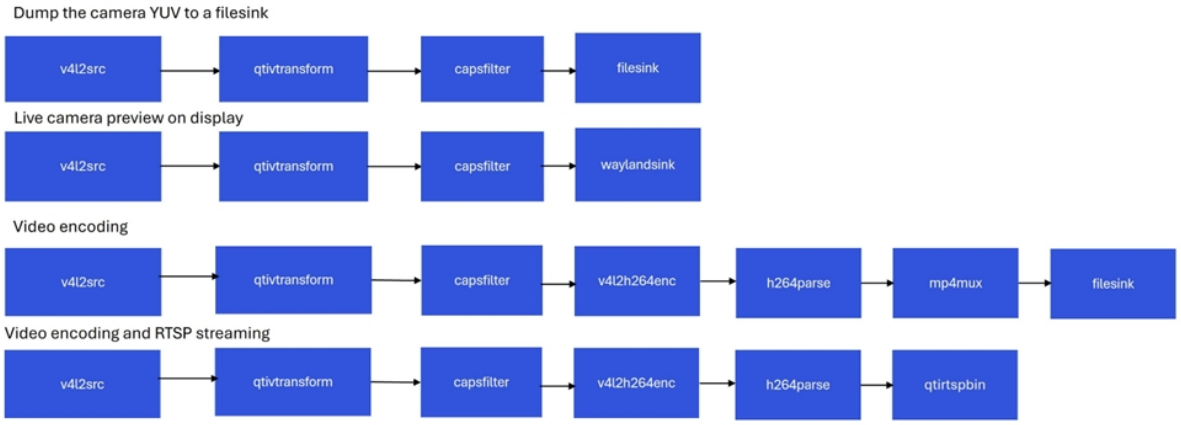


Figure 2-40 gst-usb-single-camera-app pipeline

Run the application

Before running the application, complete the Prerequisites.

To run this sample application without the AI inferencing, do the following:

1. Use the `config-usb-camera-app.json` file from AI USB camera.
2. In the `config-usb-camera-app.json` file, set `enable-object-detection` to `FALSE`.
3. Push the file to the `/etc/configs/` directory on the target device.
4. Run the gst-ai-usb-camera-app application:

   ```
   gst-ai-usb-camera-app --od-config-file=/etc/config/config-usb-camera-app.json
   ```

To display the available help options, run the following command in the SSH shell:

```
gst-ai-usb-camera-app -h
```

To stop the use case, use **CTRL + C**.

Expected output

The table lists the expected output for each use case:

Table 2-32 Expected output for gst-ai-usb-single-camera-app–without AI

Use case	Output
Preview	Preview of the camera stream. See the Figure 2-41.
Save the MP4 and YUV data	Save the output to a file at `/etc/` folder path.
RTSP	View the output on a remote display.

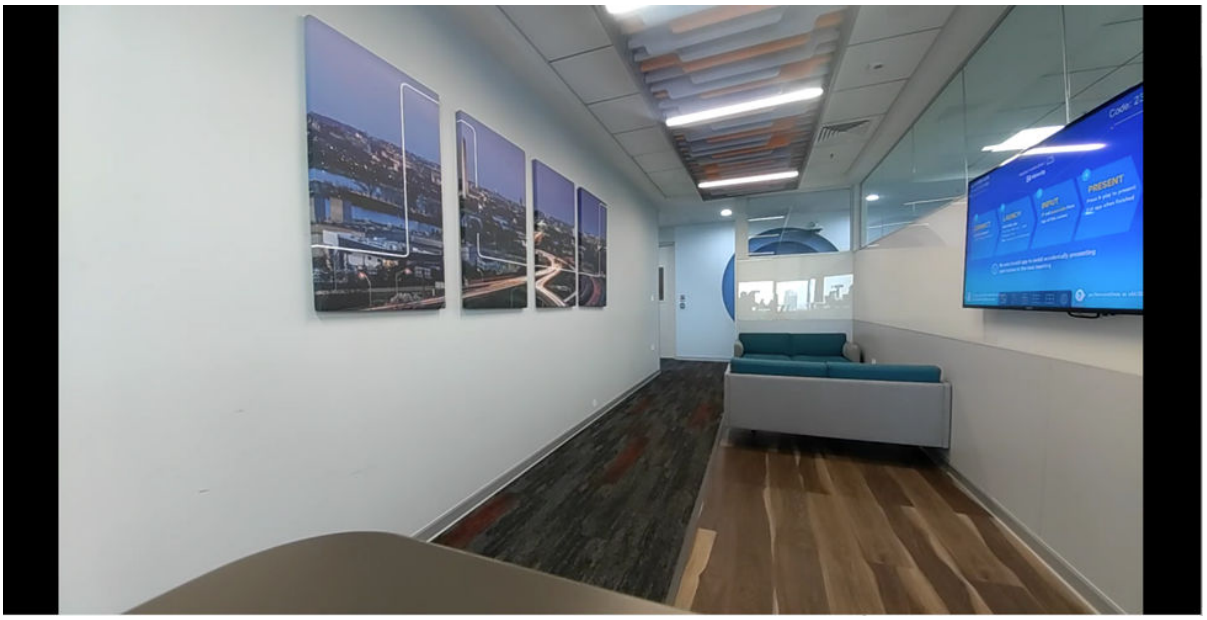


Figure 2-41 Expected output for gst-usb-single-camera-app application - Preview

Pipeline flow

The table lists the plugins used in the USB camera streaming pipeline:

Pipeline	Description
Dump the camera YUV to a filesink	1. v4l2src captures the camera live stream. 2. qtivtransform transforms the stream data. 3. Capsfilter is applied to enforce constraints on the raw video data. 4. Filesink is used to dump the YUV data into a file.
Preview on display	1. v4l2src captures the camera live stream. 2. qtivtransform transforms the stream data. 3. Capsfilter is applied to enforce constraints on the raw video data. 4. The data is sent to the Wayland display sink (Waylandsink) for live preview.
Video encoding	1. v4l2src captures the camera live stream. 2. qtivtransform transforms the stream data. 3. Capsfilter is applied to enforce constraints on the raw video data. 4. v4l2h264enc is used to encode the video using the H.264 format. 5. H264parse is used to parse the video. 6. Mp4mux is used to multiplex the video into an MP4 container. 7. Filesink is used to write the video to a file.
Video encoding and RTSP streaming	1. qtiqmmfsrc captures the camera live stream. 2. qtivtransform transforms the stream data. 3. Capsfilter is applied to enforce constraints on the raw video data. 4. v4l2h264enc is used to encode the video using the H.264 format. 5. H264parse is used to parse the video. 6. qtirtspbin is used to load the stream to RTSP.

2.2.1.3 Multi-camera streaming

The **gst-multi-camera-example** application allows you to stream from two camera sensors simultaneously. The application composes the camera feeds side by side to display on a screen or encodes and stores the video streams to files.

NOTE This sample application isn't supported on QCS8275 and QCS9075.

A few use cases that need many camera inputs are dash camera or stereo camera. You can use this application as a reference and build your use case. For example, a security system where the goal is to capture a video from several cameras. You can either view it or archive it for a future review.

The figure shows the pipeline for several camera streams. For more information about the elements used in this pipeline, see Pipeline flow.

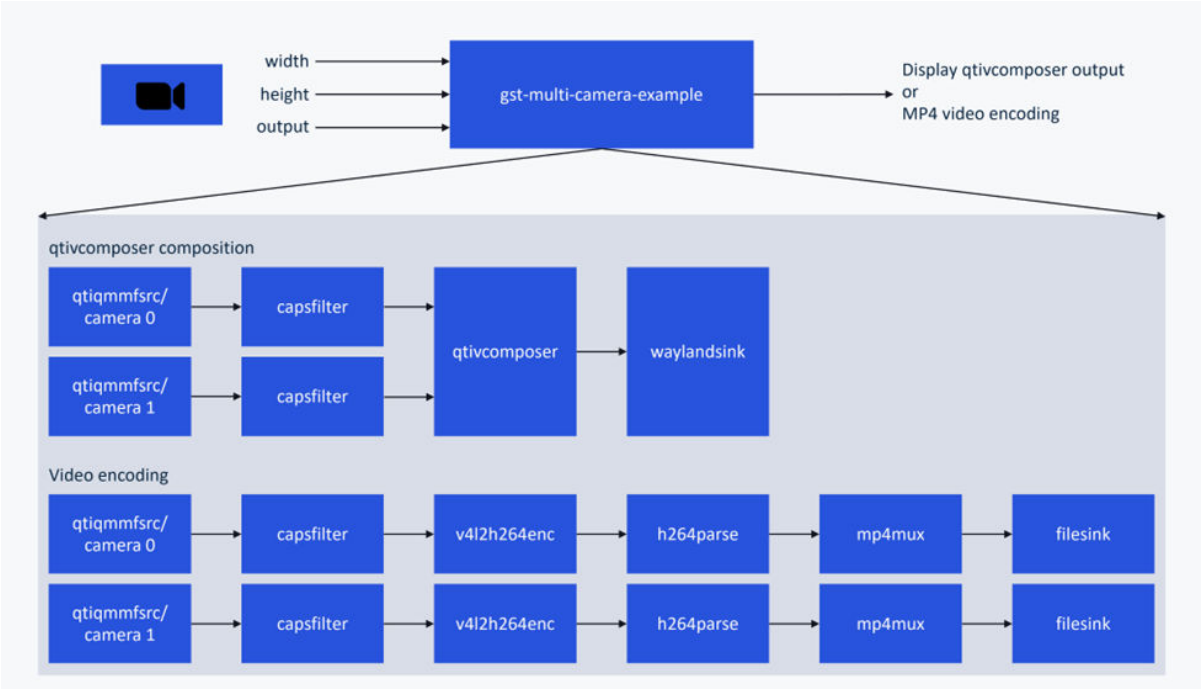


Figure 2-42 gst-multi-camera-example pipeline

Prerequisites

- If not already done so, Download and install eSDK.
- To access your host device, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is enabled, you can skip this step.
- Push the files from the Linux host computer.

  ```
  scp <filename> root@<IP address of target device>:/etc/
  ```
- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.
- Enable the display:

  ```
  export XDG_RUNTIME_DIR=/dev/socket/weston && export
  WAYLAND_DISPLAY=wayland-1
  ```
- Connect to the speaker to enable the audio. For instructions, see Set up audio hardware.

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

Run the application

NOTE In the following commands, provide the necessary parameters, such as width, height, and output type. The width and height changes are applicable to the primary camera only.

- View the Waylandsink output:

  ```
  gst-multi-camera-example -o 0 -w 1920 -h 1080
  ```

- View the encoder output:

  ```
  gst-multi-camera-example -o 1 -w 1920 -h 1080
  ```

To stop the use case, use **CTRL + C**.

To display the available help options, run the following command in the SSH shell:

```
gst-multi-camera-example --help
```

Expected output

The output is displayed on the screen and saved to a file.

- If the output type is display, you can preview the stream on the display.
- If the output type is video encoding, then the encoded files are saved at `/opt/cam1_vid.mp4` and `/opt/cam2_vid.mp4` for camera 1 and camera 2 respectively.

  Pull the files from the host computer:

  ```
  scp root@<IP address of target device>:/opt/<destination directory>
  ```

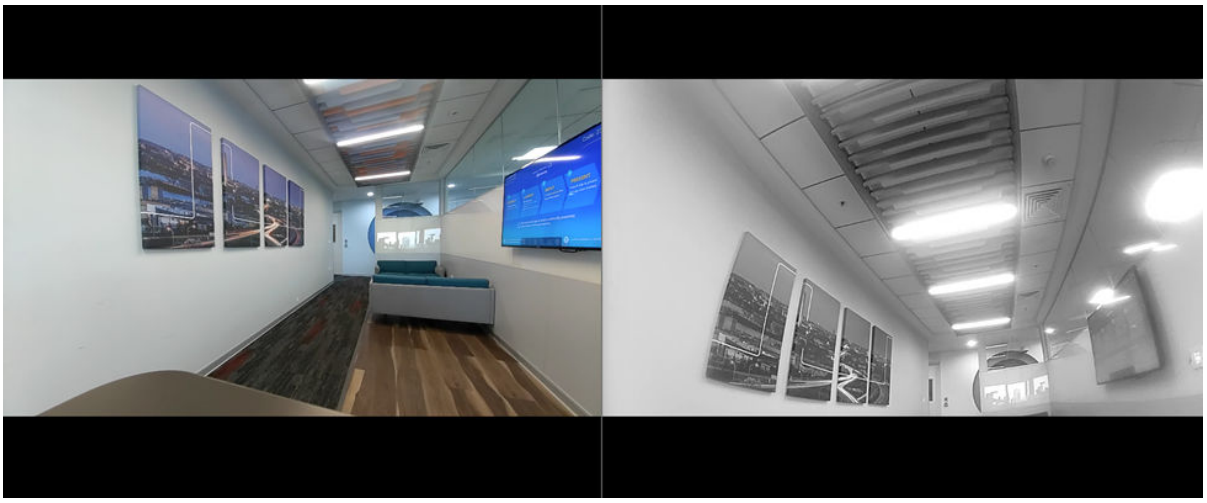


Figure 2-43 Expected output for gst-multi-camera-example application–Preview

Pipeline flow

The table lists the plugins used in the multiple camera streaming pipeline:

Pipeline	Description
Preview on display	1. qtiqmmfsrc captures video from both camera0 and camera1. 2. Capsfilter is applied to enforce constraints on the raw video data. 3. qtivcomposer composites the video streams and sends the composited video data to Wayland display sink. 4. Waylandsink shows the live preview.
Encoder dump on the device	1. qtiqmmfsrc captures video from both camera0 and camera1. 2. Capsfilter is applied to enforce constraints on the raw video data.

Pipeline	Description
	3. v4l2h264enc is used to encode the video using the H.264 format. 4. H264parse is used to parse the video. 5. Mp4mux is used to multiplex the video into an MP4 container. 6. Filesink is used to write the video to a file.

RELATED INFORMATION

"Multi-camera streaming using Python" on page 181
"Multi-camera/Multi-client use cases" on page 297

2.2.1.4 Multistream

The **gst-multi-stream-example** application allows you to access several streams from a single camera and display them simultaneously on a screen or save them in an MP4 format.

NOTE This sample application isn't supported on QCS8275 and QCS9075.

For example, you can use this application for a camera security system where the video is captured from a camera. This video can be simultaneously streamed over a network, displayed on a monitor, and archived for future review.

The figure shows the pipeline, which is run with flags to specify width, height, stream count, and output file.

For more information, see Pipeline flow.

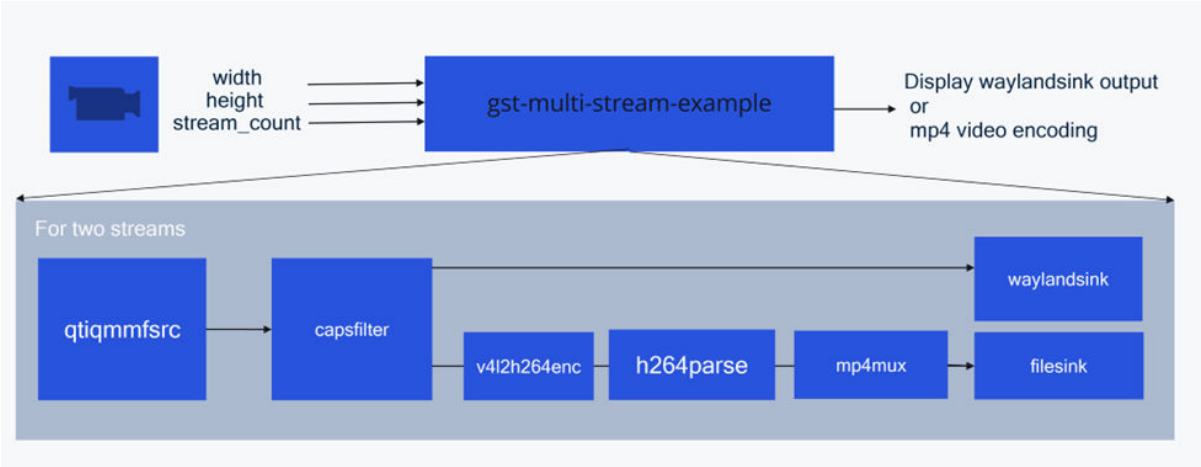


Figure 2-44 gst-multi-stream-example pipeline

Prerequisites

- If not already done so, Download and install eSDK.
- To access your host device, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is enabled, you can skip this step.
- Push the files from the Linux host computer.

  ```
  scp <filename> root@<IP address of target device>:/etc/
  ```
- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.

- Enable the display:

  ```
  export XDG_RUNTIME_DIR=/dev/socket/weston && export
  WAYLAND_DISPLAY=wayland-1
  ```

- Connect to the speaker to enable the audio. For instructions, see Set up audio hardware.

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

Run the application

- Run the use case:

  ```
  gst-multi-stream-example -w 1920 -h 1080 -n 2
  ```

- To display the available help options, run the following command in the SSH shell:

  ```
  gst-multi-stream-example --help
  ```

- To stop the use case, use **CTRL + C**.

Expected output

The output streams are displayed simultaneously on the screen or saved to a file.

Pull the files from the host computer:

```
scp root@<IP address of target device>:/opt/<destination directory>
```

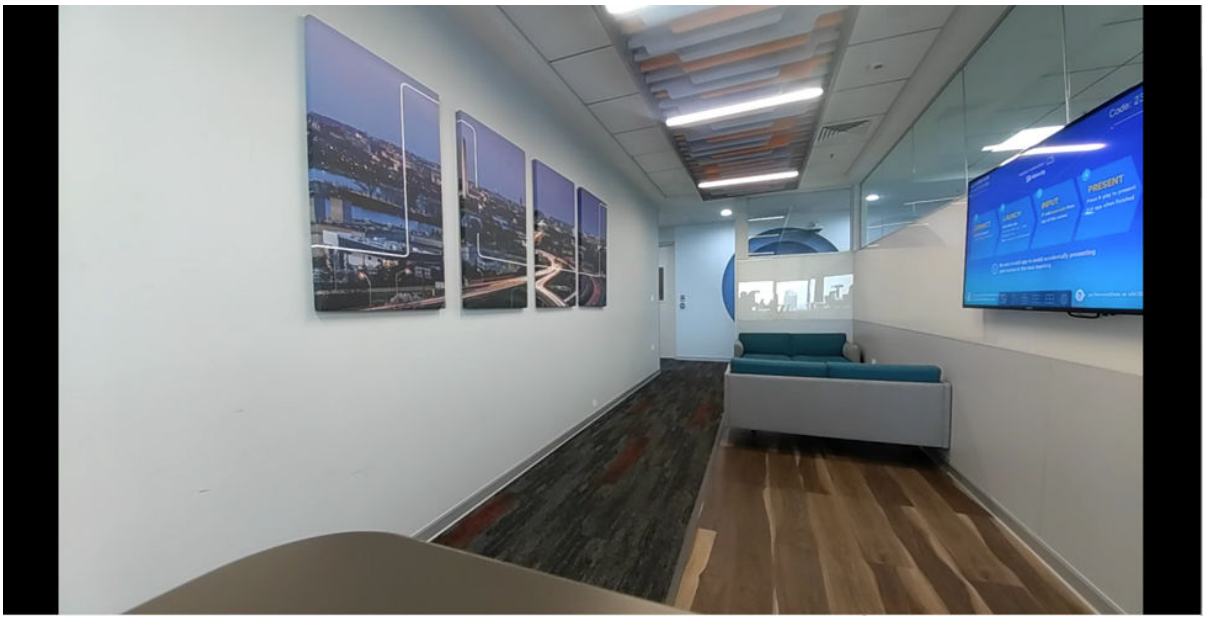


Figure 2-45 Expected output for gst-multi-stream-example application - Preview

Pipeline flow

The table lists the plugins used in the multistream pipeline:

Plugin	Description
qtiqmmfsrc	Captures the video stream and feeds it into the capsfilter.
Capsfilter	Branches the source into two separate pipelines. In one pipeline, the output is sent to Waylandsink for display.
	■ In the other pipeline, the video undergoes the following: a. Encoded using v4l2h264enc. b. Parsed with h264parse. c. Multiplexed into an MP4 format with mp4mux. d. Written to a file using filesink.

Known issue

The default bit rate of the encoded video is set to low, resulting in suboptimal quality for higher resolutions.

2.2.1.5 Live snapshot

The **gst-snapshot-stream-example** application allows you to create two streams, a preview stream and a snapshot stream. The preview stream is displayed while the snapshot stream captures snapshots and saves them to a file in JPEG format.

NOTE This sample application isn't supported on QCS8275 and QCS9075.

For the snapshot stream, you can determine the number of snapshots.

The figure shows the pipeline, which allows for simultaneous live preview and snapshot capture from the same video source.

For more information, see Pipeline flow.

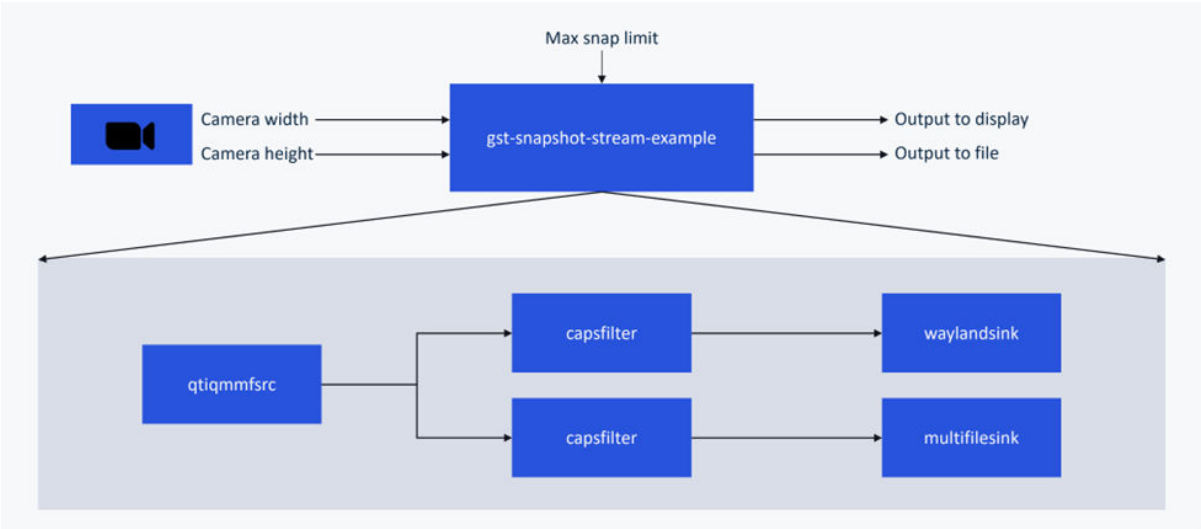


Figure 2-46 gst-snapshot-stream-example pipeline

Prerequisites

- If not already done so, Download and install eSDK.
- To access your host device, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is enabled, you can skip this step.
- Push the files from the Linux host computer.

  ```
  scp <filename> root@<IP address of target device>:/etc/
  ```
- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.
- Enable the display:

  ```
  export XDG_RUNTIME_DIR=/dev/socket/weston && export
  WAYLAND_DISPLAY=wayland-1
  ```
- Connect to the speaker to enable the audio. For instructions, see Set up audio hardware.

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

Run the application

- Run the use case. Specify the application with parameters, such as camera width, height, and the maximum snapshot count for dumping JPEG files.

  ```
  gst-snapshot-stream-example -W 1280 -H 720 -w 3840 -h 2160 -c 5
  ```
- To display the available help options, run the following command in the SSH shell:

  ```
  gst-snapshot-stream-example --help
  ```
- To stop the use case, use **CTRL + C**.

Expected output

The application generates an output for preview on the display and dumps the JPEG files in the respective location on the device.

After the use case stops, the snapshot output files are available at the `/opt/` path.

Pipeline flow

The table lists the plugins used in the live snapshot pipeline:

Plugin	Description
qtiqmmfsrc	Captures the video stream and feeds it into the capsfilter.
Capsfilter	Branches the source into two separate pipelines. In one pipeline, the constraints on the raw video data are enforced, and then the video data is sent to Waylandsink for live preview.
	1. In the other pipeline, the frames are written to multiple files using multi filesink. 2. The snapshots or frames from the video stream are saved to a disk.

2.2.1.6 Dynamic configuration

A common scenario in Edge AI applications is changing the image resolution at runtime for minimal frame loss or gap in the display, or adding and removing several streams for camera reconfiguration. Dynamic configuration sample applications allow you to perform these changes.

Activate-deactivate streams at runtime

The **gst-activate-deactivate-streams-runtime-example** application allows you to change the resolution dynamically without the need to reconfigure the camera sensor.

NOTE This sample application isn't supported on QCS8275 and QCS9075.

This application can be used in surveillance systems, videoconferencing, and broadcasting where several live streams are processed in parallel and you can control the active streams dynamically.

The figure shows the pipeline, where the application creates two streams of different resolutions, and activates and deactivates them at runtime. For information about the plugins used in this pipeline, see Pipeline flow.

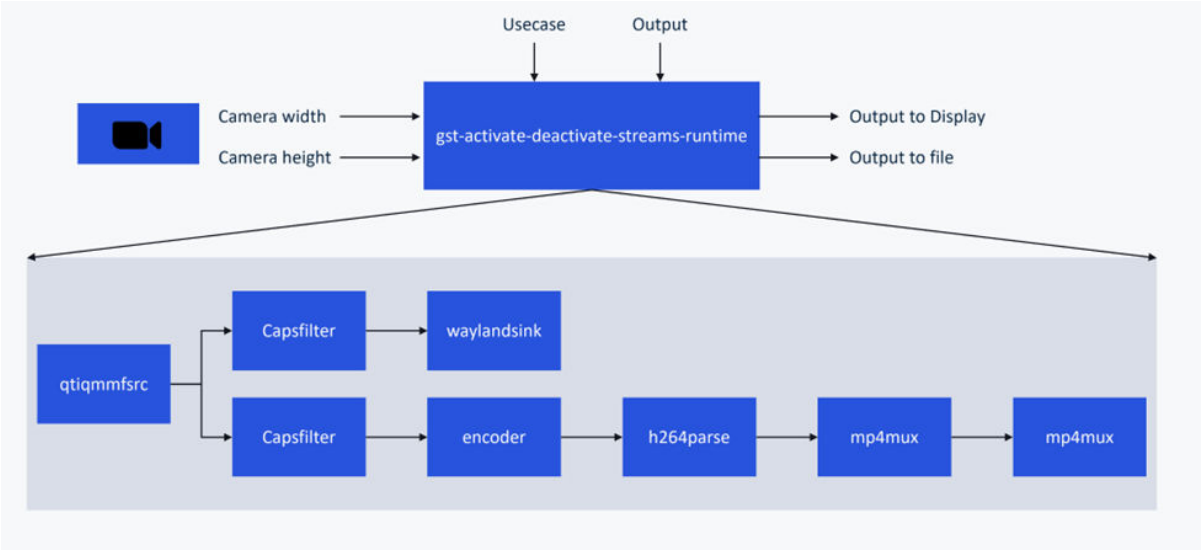


Figure 2-47 gst-activate-deactivate-streams-runtime-example pipeline

Prerequisites

- If not already done so, Download and install eSDK.
- To access your host device, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is enabled, you can skip this step.
- Push the files from the Linux host computer.

  ```
  scp <filename> root@<IP address of target device>:/etc/
  ```
- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.

- Enable the display:

```
export XDG_RUNTIME_DIR=/dev/socket/weston && export
WAYLAND_DISPLAY=wayland-1
```

- Connect to the speaker to enable the audio. For instructions, see Set up audio hardware.

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

Run the application

The table lists the application use case modes. The NULL state is the initial state of a GStreamer element.

NOTE Currently, the application is compatible only with the Basic mode. Full mode will be supported in a future release.

Use case mode	Description
Basic	Activates or deactivates several streams *without entering* the NULL state.
Full	Activates or deactivates several streams *by entering* the NULL state.

For more information on GStreamer states, seehttps://gstreamer.freedesktop.org/documentation/additional/design/states.html?gi-language=c

The following are the commands to run the use cases. Ensure that you configure the use case and output type parameters.

Basic use cases

- Stream on Waylandsink:

```
gst-activate-deactivate-streams-runtime-example -u Basic -o Display
```

- Encode to MP4 file:

```
gst-activate-deactivate-streams-runtime-example -u Basic -o File
```

Full use cases

- Stream on Waylandsink:

```
gst-activate-deactivate-streams-runtime-example -u Full -o Display
```

- Encode to MP4 file:

```
gst-activate-deactivate-streams-runtime-example -u Full -o File
```

Additionally, you can do the following:

- Pull the files from the host computer:

```
scp root@<IP address of target device>:/opt/<destination directory>
```

- Display the available help options:

```
gst-activate-deactivate-streams-runtime-example --help
```

To stop the use case, use **CTRL + C**.

Expected output

During the execution, the first stream is activated followed by the second stream. After a while, the streams deactivate.

- If the output option is set to `Display`, the output is displayed as a preview.
- If the output option is set to `File`, the encoded video files are generated.

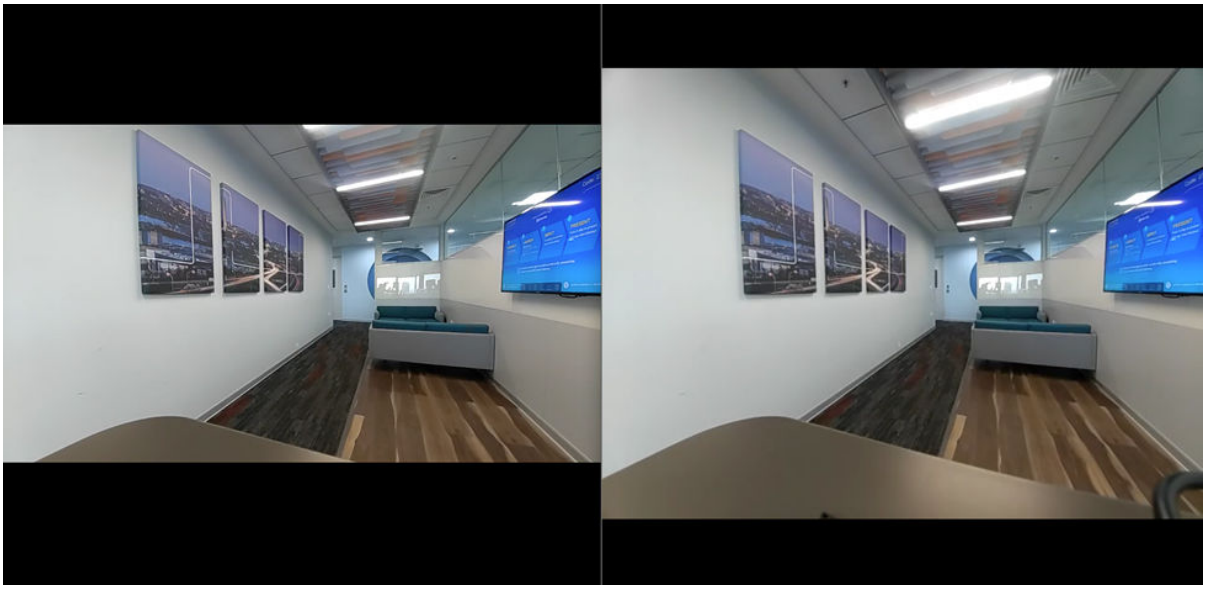


Figure 2-48 Expected output for gst-activate-deactivate-streams-runtime-example application

Pipeline flow

The table lists the plugins used in the gst-activate-deactivate-streams-runtime-example pipeline:

Plugin	Description
qtiqmmfsrc	Captures the camera feed and sends it into the capsfilter.
Capsfilter	Passes the feed to Waylandsink for display.
	Passes the feed to filesink to save it to a file.

Add-remove streams at runtime

The **gst-add-remove-streams-runtime** application allows you to add or remove several camera streams dynamically during runtime.

NOTE This sample application isn't supported on QCS8275 and QCS9075.

The figure shows the pipeline, which demonstrates the ability of the application to add or remove the streams at runtime with camera reconfiguration. It creates three streams and adds or removes them in different order.

For information about the plugins used in the pipeline, see Pipeline flow.

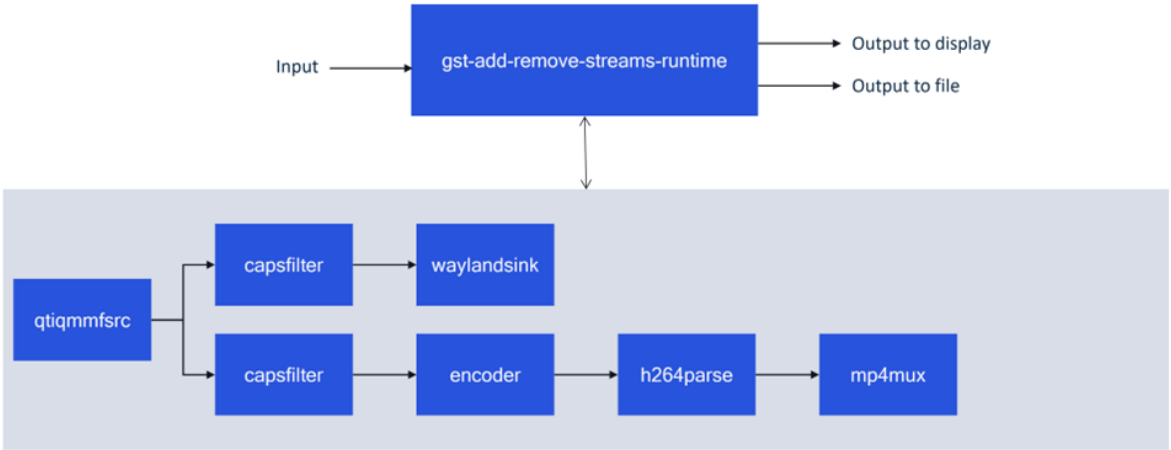


Figure 2-49 gst-add-remove-streams-runtime pipeline

Prerequisites

- If not already done so, Download and install eSDK.
- To access your host device, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is enabled, you can skip this step.
- Push the files from the Linux host computer.

  ```
  scp <filename> root@<IP address of target device>:/etc/
  ```
- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.
- Enable the display:

  ```
  export XDG_RUNTIME_DIR=/dev/socket/weston && export
  WAYLAND_DISPLAY=wayland-1
  ```
- Connect to the speaker to enable the audio. For instructions, see Set up audio hardware.

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

Run the application

Verify if many streams can run simultaneously:

```
gst-add-remove-streams-runtime <OPTION>
```

The following list provides the commands to run the application for various use cases:

- Default mode preview on the display:

  ```
  gst-add-remove-streams-runtime
  ```
- Preview the stream:

  ```
  gst-add-remove-streams-runtime -o Display
  ```

- Encode the stream:

  ```
  gst-add-remove-streams-runtime -o File
  ```

- Pull the files from the host computer:

  ```
  scp root@<IP address of target device>:/opt/ <destination directory>
  ```

- Display the available help options:

  ```
  gst-add-remove-streams-runtime --help
  ```

To stop the use case, use **CTRL + C**.

Expected output

- If you choose to preview, the stream is displayed on the screen.
- If you choose to encode the stream, the encoded stream is saved at `/opt/video_%d.mp4`.

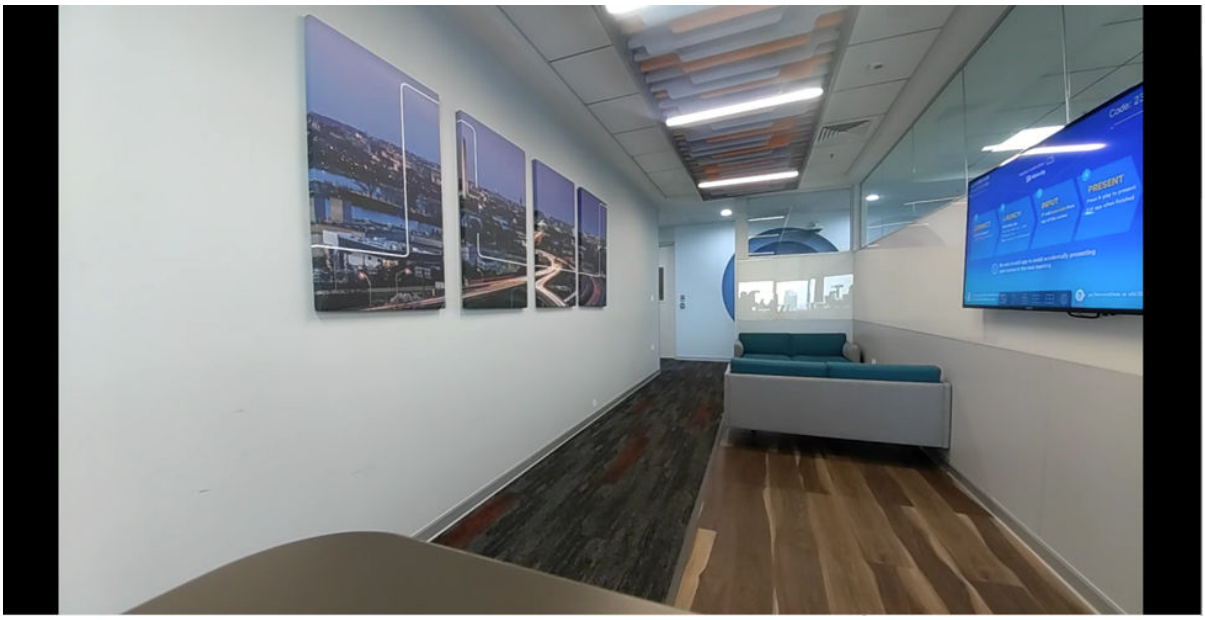


Figure 2-50 Expected output for gst-add-remove-streams-runtime application–Preview

Pipeline flow

The table lists the plugins used in the gst-add-remove-streams-runtime pipeline:

Plugin	Description
qtiqmmfsrc	Captures the video stream and feeds it into the capsfilter.
Capsfilter	Branches the source into two separate pipelines. In one pipeline, the output is sent to Waylandsink for display.
	▪ In the other pipeline, the video undergoes the following: a. Encoded using v4l2h264enc. b. Parsed with h264parse. c. Multiplexed into an MP4 format with mp4mux.

2.2.1.7 Camera frame processing

The **gst-appsink-example** application allows you to extract the buffer samples from a camera pipeline. These buffers can configure the camera resolution with the options such as width and height.

For example, the `appsink` plugin can be used for a multimedia application that processes the video frame samples in real-time. Filters or transformations can be applied to the data before playing it back.

The figure shows the pipeline, which configures the camera resolution using parameters such as width and height. When appsink emits a signal indicating the availability of a new sample in the pipeline, the application extracts the buffer from this sample.

For information on the plugins used in this pipeline, see Pipeline flow.

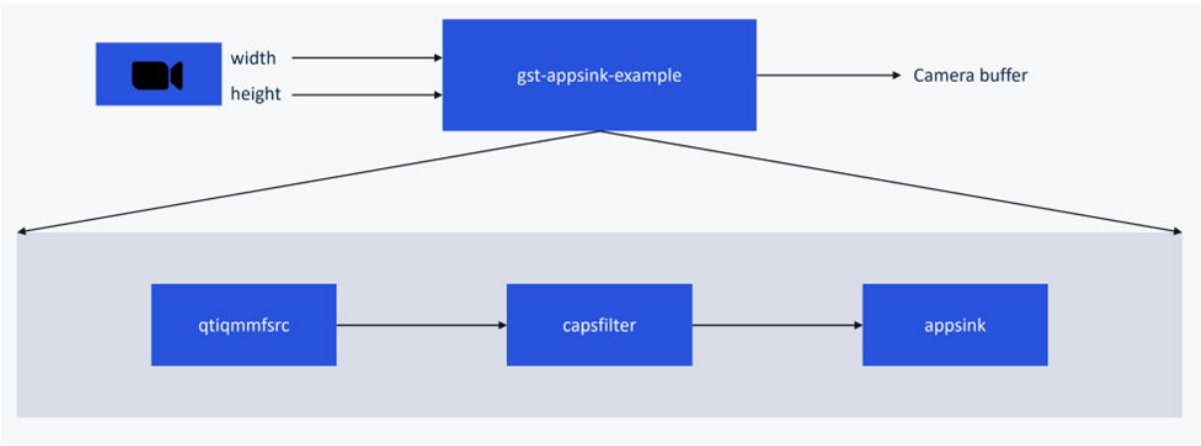


Figure 2-51 gst-appsink-example pipeline

Prerequisites

- If not already done so, Download and install eSDK.
- To access your host device, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is enabled, you can skip this step.
- Push the files from the Linux host computer.

  ```
  scp <filename> root@<IP address of target device>:/etc/
  ```
- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.
- Enable the display:

  ```
  export XDG_RUNTIME_DIR=/dev/socket/weston && export
  WAYLAND_DISPLAY=wayland-1
  ```
- Connect to the speaker to enable the audio. For instructions, see Set up audio hardware.

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

Run the application

```
gst-appsink-example -w 1920 -h 1080
```

- To display the available help options, run the following command in the SSH shell:

  ```
  gst-appsink-example --help
  ```

- To stop the use case, use **CTRL + C**.

Expected output

The user interface displays the following message:

```
Received a buffer, doing some processing.
```

```
sh-5.1# gst-appsink-example -w 1920 -h 1080
GBM_INFO::msmgbm_mapper(262)::gbm mapper instantiated

gbm_create_device(224): Info: backend name is: msm_drm

 Linking appsink elements ..

 All elements are linked successfully

 Setting pipeline to PAUSED state ...

 Pipeline is live and does not need PREROLL.

 Application is running...

Pipeline state changed from NULL to READY:

Pipeline state changed from READY to PAUSED:
GBM_ERR::msmgbm_bo_import_gbm_buf(1605)::DRM_IOCTL_PRIME_FD_TO_HANDLE

GBM_ERR::msmgbm_bo_import_gbm_buf(1633)::DRM_IOCTL_PRIME_FD_TO_HANDLE


Pipeline state changed from PAUSED to PLAYING:
GBM_ERR::msmgbm_bo_import_gbm_buf(1605)::DRM_IOCTL_PRIME_FD_TO_HANDLE

GBM_ERR::msmgbm_bo_import_gbm_buf(1633)::DRM_IOCTL_PRIME_FD_TO_HANDLE


 Received a buffer, doing some processing ...
```

Figure 2-52 Expected output for gst-appsink-example application

Pipeline flow

The table lists the plugins used in the camera frame processing pipeline:

Plugin	Description
qtiqmmfsrc	Captures the camera feed and sends it into the capsfilter.
Capsfilter	Ensures that the video stream is in the correct format.
Appsink	Provides the video data as an output to an application.

2.2.1.8 Camera metadata

The **gst-camera-metadata-example** application starts with a menu that allows you to choose between multiple properties of the camera plugin such as `video-metadata`, `static-metadata`, and `image-metadata`.

NOTE This sample application isn't supported on QCS8275 and QCS9075.

A practical use case for this application is handling camera metadata such as white balance, International Organization for Standardization (ISO) sensitivity, and exposure to tune the camera.

The figure shows the execution of appsink when a new sample is available in the pipeline. The application is run from the command-line with specific options for custom pipeline, display type, and timestamp location.

For information about the plugins used in this pipeline, see Pipeline flow.

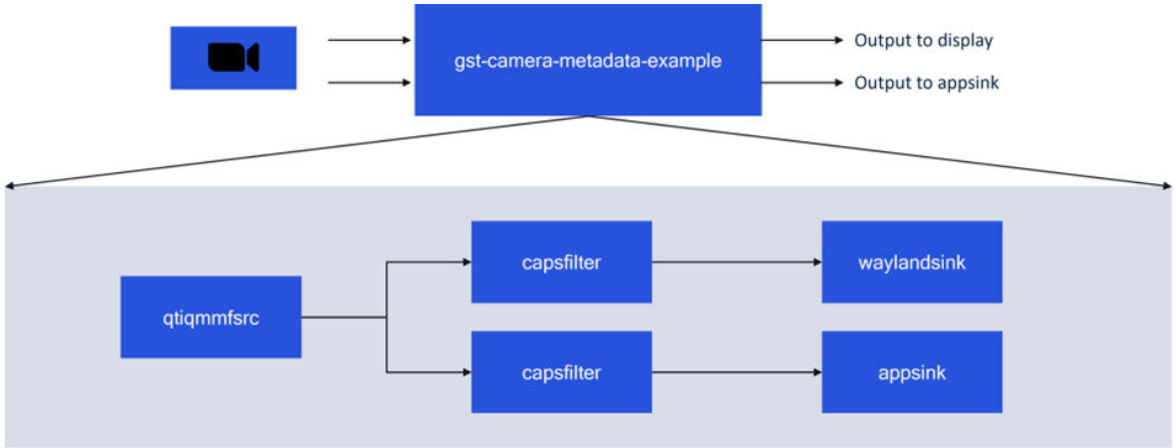


Figure 2-53 gst-camera-metadata-example pipeline

Prerequisites

- If not already done so, Download and install eSDK.
- To access your host device, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is enabled, you can skip this step.
- Push the files from the Linux host computer.

  ```
  scp <filename> root@<IP address of target device>:/etc/
  ```
- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.

- Enable the display:

  ```
  export XDG_RUNTIME_DIR=/dev/socket/weston && export
  WAYLAND_DISPLAY=wayland-1
  ```

- Connect to the speaker to enable the audio. For instructions, see Set up audio hardware.

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

Run the application

- Run the use case:

  ```
  gst-camera-metadata-example -d 1
  ```

- Run the custom pipeline use case:

  ```
  gst-camera-metadata-example -p "qtiqmmfsrc name=camera camera.video_0 !
  video/x-raw,format=NV12,width=1280,height=720,framerate=30/1,interlace-
  mode=progressive,colorimetry=bt601 ! queue ! appsink name=sink emit-
  signals=true enable-last-sample=false"
  ```

- To display the available help options, run the following command in the SSH shell:

  ```
  gst-camera-metadata-example --help
  ```

- To stop the use case, use **CTRL + C**.

Expected output

- During the execution, the application requires the command-line options to output the camera metadata and options available to set or get the vendor tags.
- On successful execution of the application, you can find and dump the respective metadata information.

```
---------------------------------------------------- Vendor tags ----------------------------------------------------

--- TAG ID ---      ---------------------- SECTION ----------------------      ------------------ TAG ------------------
2147614720          org.codeaurora.qcamera3.sharpness                          strength
2147680256          org.codeaurora.qcamera3.contrast                           level
2147745792          org.codeaurora.qcamera3.iso_exp_priority                   use_iso_exp_priority
2147745793          org.codeaurora.qcamera3.iso_exp_priority                   use_iso_value
2147745794          org.codeaurora.qcamera3.iso_exp_priority                   select_priority
2148139008          org.codeaurora.qcamera3.ae_bracket                         mode
2148925440          org.quic.camera2.streamconfigs                             HDRVideoMode
2148990976          org.quic.camera2.jpegquantizationtables                    OEMJPEGLumaQuantizationTable
2148990977          org.quic.camera2.jpegquantizationtables                    OEMJPEGChromaQuantizationTable
2149711872          org.quic.camera2.tuning.feature                            Feature1Mode
2149711873          org.quic.camera2.tuning.feature                            Feature2Mode
2150039552          org.quic.camera2.ref.cropsize                              RefCropSize
2150105092          org.codeaurora.qcamera3.manualWB                           partial_mwb_mode
2152005632          org.quic.camera.manualExposure                             disableFPSLimits
2152005633          org.quic.camera.manualExposure                             overrideSensorFrameDuration
2152071168          org.quic.camera.EarlyPCRenable                             EarlyPCRenable
2153381888          org.codeaurora.qcamera3.adrc                               disable
2153447424          org.codeaurora.qcamera3.sessionParameters                  availableStreamMap
2153447425          org.codeaurora.qcamera3.sessionParameters                  overrideResourceCostValidation
2153447426          org.codeaurora.qcamera3.sessionParameters                  enableMFNR
2155347968          org.quic.camera2.statsNNControl                            Enable
2156855296          org.codeaurora.qcamera3.exposure_metering                  exposure_metering_mode
2158100480          org.codeaurora.qcamera3.saturation                         use_saturation
2160132097          org.codeaurora.qcamera3.ir_led                             mode

-------------------------------------------------------------------------------------------------------------------------



------------------------- video-metadata -------------------------
   (1) List all available tags
   (2) Dump all tags values in a file
   (3) Dump custom tags values in a file
   (4) Get a tag
   (5) Set a tag
-------------------------------------------------------------
   (b) Back

Choose an option:
```

Figure 2-54 Expected output for gst-camera-metadata-example application

Pipeline flow

The table lists the plugins used in the camera metadata pipeline:

Plugin	Description
qtiqmmfsrc	Captures the camera feed and sends it into the capsfilter.
Capsfilter	Passes the feed to Waylandsink for display.
	If display option isn't provided, passes the feed to appsink.

The application extracts the buffer from the sample for further processing. During execution, the application expects an argument on which the following options should be displayed on the console.

- Video metadata
- Image metadata
- Static metadata
- Session metadata

Based on the options selected, the application prints the output on the console.

Known issues

When the **GetTag** menu option is selected, the capture options don't work.

2.2.1.9 Camera stream configuration

The **gst-add-streams-as-bundle-example** application allows you to activate and configure camera streams as a bundle instead of configuring each stream one at a time, thus, reducing the delay in configuring new streams.

NOTE This sample application isn't supported on QCS8275 and QCS9075.

The figure shows the pipeline, which takes the configuration parameters such as width and height, processes to either display on a screen, or encodes and saves to a file. For more information about the plugins used in the pipeline, see Pipeline flow.

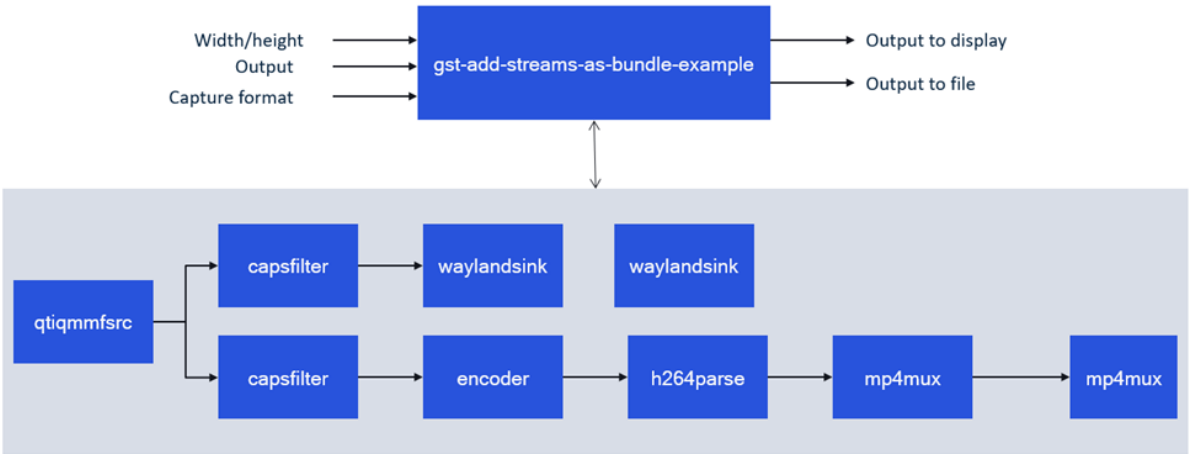


Figure 2-55 gst-add-streams-as-bundle-example pipeline

Prerequisites

- If not already done so, Download and install eSDK.
- To access your host device, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is enabled, you can skip this step.
- Push the files from the Linux host computer.

  ```
  scp <filename> root@<IP address of target device>:/etc/
  ```
- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.
- Enable the display:

  ```
  export XDG_RUNTIME_DIR=/dev/socket/weston && export
  WAYLAND_DISPLAY=wayland-1
  ```
- Connect to the speaker to enable the audio. For instructions, see Set up audio hardware.

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

Run the application

To run the application, use the following syntax:

```
gst-add-streams-as-bundle-example <OPTION>
```

The following are the commands to run the application for various use cases:

- Preview the stream:

  ```
  gst-add-streams-as-bundle-example -o Display
  ```

- Encode the stream and save it to a file:

  ```
  gst-add-streams-as-bundle-example -o File
  ```

  The encoded stream is saved at `/opt/video_%d.mp4`.

  Pull the files from the host computer:

  ```
  scp root@<IP address of target device>:/opt/<destination directory>
  ```

The application stops after the execution is completed.

- To display the available help options, run the following command in the SSH shell:

  ```
  gst-add-streams-as-bundle-example --help
  ```

- To stop the use case, use **CTRL + C**.

Expected output

One stream is sent to display and the other stream is saved to a file.

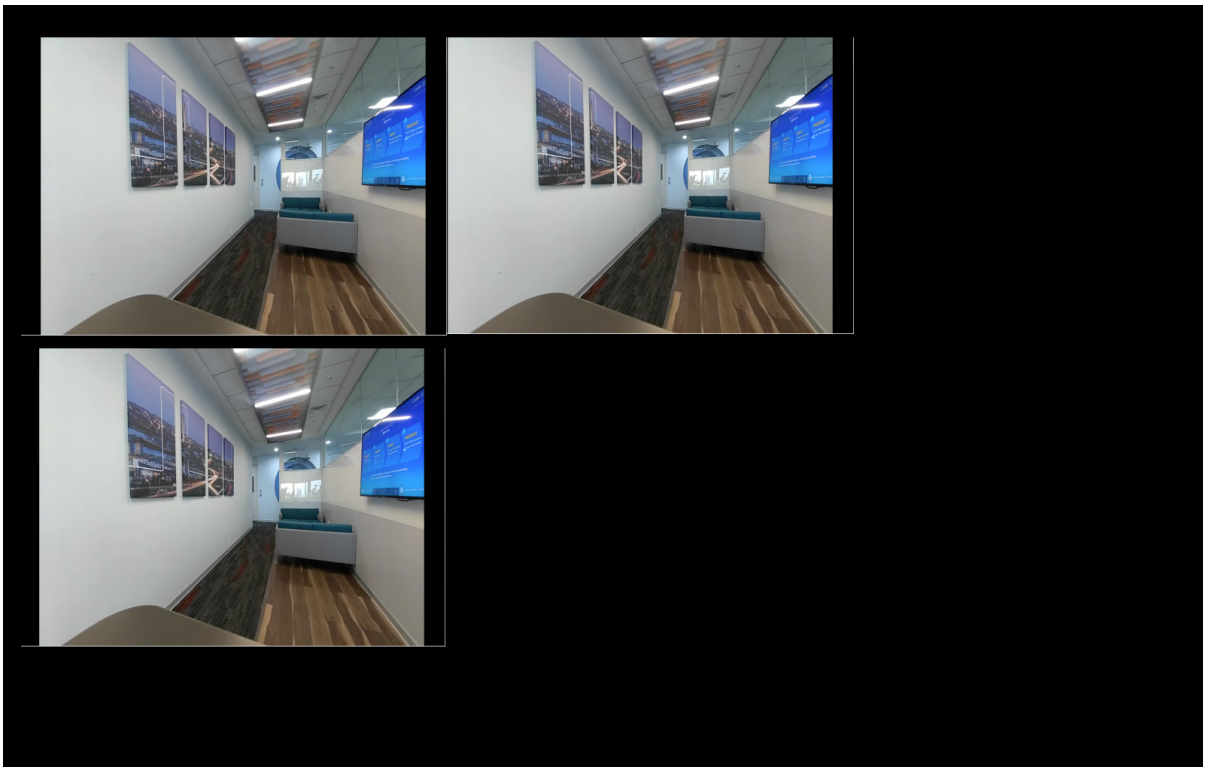


Figure 2-56 Expected output for gst-add-streams-as-bundle-example application

Pipeline flow

The table lists the plugins used in the camera stream configuration pipeline:

Plugin	Description
qtiqmmfsrc	Captures the video stream and feeds it into the capsfilter.
Capsfilter	Branches the source into two separate pipelines. In one pipeline, the output is sent to Waylandsink for display.
	▪ In the other pipeline, the video undergoes the following: a. Encoded using v4l2h264enc. b. Parsed with h264parse. c. Multiplexed into an MP4 format with mp4mux. d. Written to a file using filesink.

2.2.1.10 Camera burst capture

The **gst-camera-burst-capture-example** application allows you to capture snapshots in the Burst mode. It captures five burst snapshots delayed by a 10 second timer and displays the snapshots on a screen.

NOTE This sample application isn't supported on QCS8275 and QCS9075.

The figure shows the pipeline, which captures the snapshots and processes them in using one of the following three branches:

- They're displayed on the screen.
- They're encoded and saved to a file.
- They're saved to a buffer and based on the capture format (BAYER/RAW/JPEG), saved to the device storage.

For more information about the plugins used in the pipeline, see Pipeline flow.

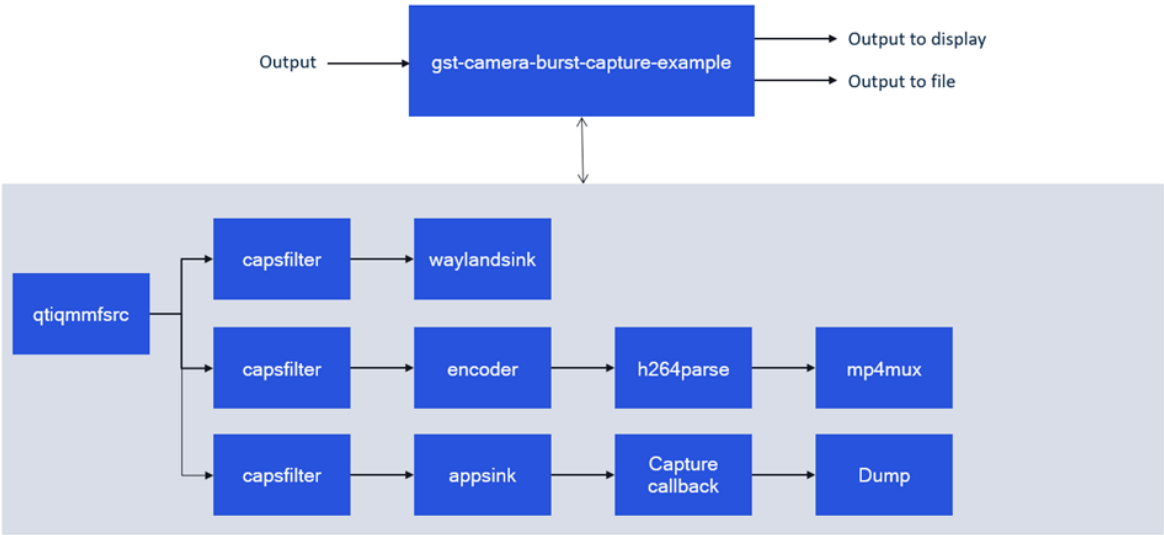


Figure 2-57 gst-camera-burst-capture-example pipeline

Prerequisites

- If not already done so, Download and install eSDK.
- To access your host device, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is enabled, you can skip this step.
- Push the files from the Linux host computer.

  ```
  scp <filename> root@<IP address of target device>:/etc/
  ```
- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.
- Enable the display:

  ```
  export XDG_RUNTIME_DIR=/dev/socket/weston && export
  WAYLAND_DISPLAY=wayland-1
  ```
- Connect to the speaker to enable the audio. For instructions, see Set up audio hardware.

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

Run the application

- Display stream and JPEG capture:

  ```
  gst-camera-burst-capture-example -w 1280 -h 720 -p 1 -c 0
  ```
- Encode stream and JPEG capture:

  ```
  gst-camera-burst-capture-example -w 1280 -h 720 -p 0 -c 0
  ```

To display the available help options, run the following command in the SSH shell:

```
gst-camera-burst-capture-example --help
```

To stop the use case, use **CTRL + C**.

Expected output

- If you choose to display the stream, the snapshots are displayed on the screen.
- If you choose to encode the stream, the encoded stream is saved at `/opt/video_%d.mp4`.

  Pull the files from host computer:

  ```
  scp root@<IP address of target device>:/opt/<destination directory>
  ```

Pipeline flow

The table lists the plugins used in the camera burst capture pipeline:

Pipeline	Description
Display preview	1. qtiqmmfsrc captures the camera live stream and uses capsfilter to create a branch for preview. 2. The data is sent to the Wayland display sink (Waylandsink) for preview.
Encode and save to a file	1. qtiqmmfsrc captures the camera live stream and uses capsfilter to create a branch for encoding. 2. The snapshots are encoded.

Pipeline	Description
	3. H264parse is used to parse the captured data. 4. Mp4mux is used to multiplex the data.
BAYER/RAW/JPEG dump	1. qtiqmmfsrc captures the camera live stream and uses capsfilter to create a branch for preview. 2. Appsink is used to capture callback and save the snapshots in the storage dump.

Known issue:

The test cases for 144p and 240p resolutions are failing.

2.2.1.11 Camera switch

The **gst-camera-switch-example** application allows you to switch between two cameras.

NOTE This sample application isn't supported on QCS8275 and QCS9075.

The figure shows the pipeline, which demonstrates the camera switch between the two cameras of the device without changing the state of the pipeline. The switching is done in the Playing state after every 5 seconds.

For more information about the plugins used in this pipeline, see Pipeline flow.

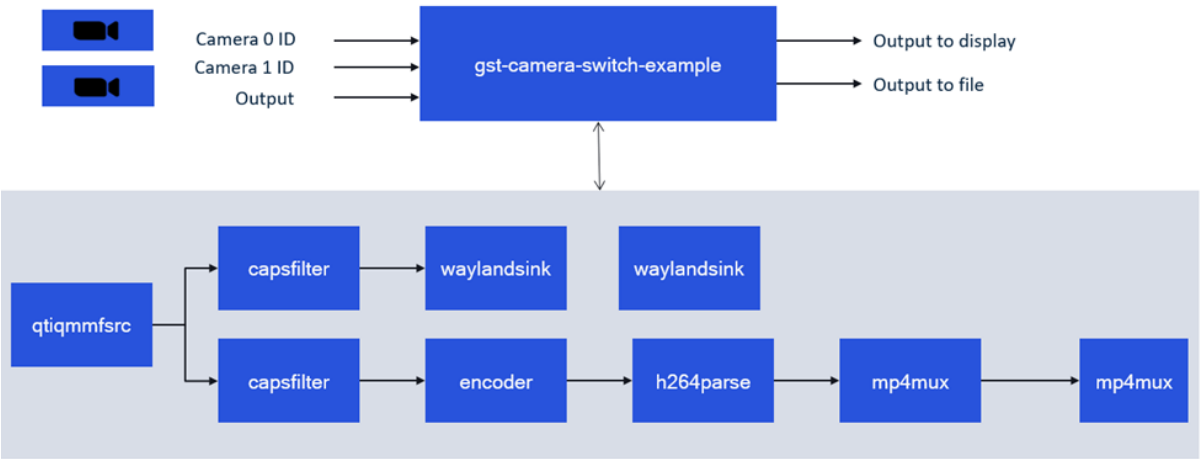


Figure 2-58 gst-camera-switch-example pipeline

Prerequisites

- If not already done so, Download and install eSDK.
- To access your host device, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is enabled, you can skip this step.
- Push the files from the Linux host computer.

  ```
  scp <filename> root@<IP address of target device>:/etc/
  ```
- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.

- Enable the display:

  ```
  export XDG_RUNTIME_DIR=/dev/socket/weston && export
  WAYLAND_DISPLAY=wayland-1
  ```

- Connect to the speaker to enable the audio. For instructions, see Set up audio hardware.

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

Run the application

To run the application, use the following syntax:

```
gst-camera-switch-example <OPTION>
```

- Encode the stream after switching (default option):

  ```
  gst-camera-switch-example
  ```

- Display the stream after switching:

  ```
  gst-camera-switch-example -d
  ```

To display the available help options, run the following command in the SSH shell:

```
gst-camera-switch-example --help
```

To stop the use case, use **CTRL + C**.

Expected output

The application generates the output as a preview or an encoded MP4 file.

Pull the files from the host computer:

```
scp root@<IP address of target device>:/opt/<destination directory>
```

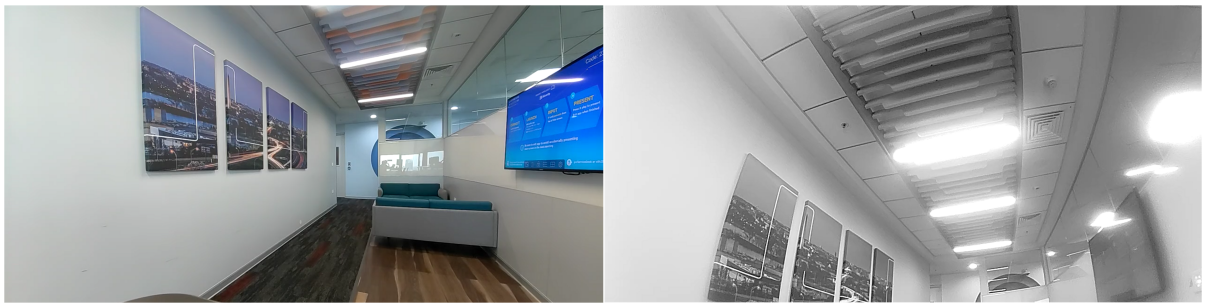


Figure 2-59 Expected output for gst-camera-switch-example application

Pipeline flow

The table lists the plugins used in the camera switch pipeline:

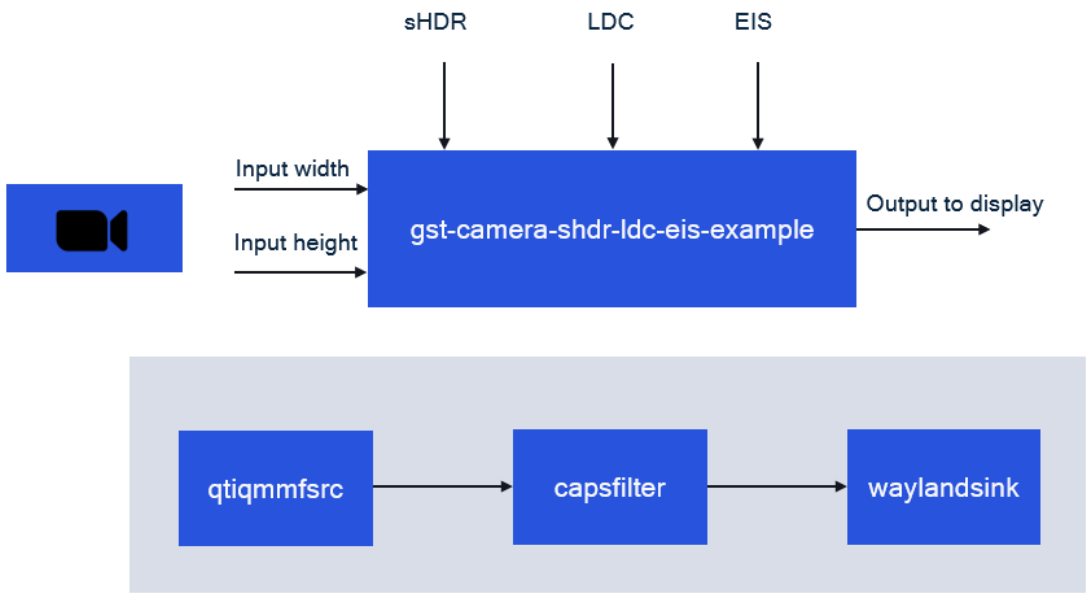
Pipeline	Description
Display preview	1. qtiqmmfsrc captures the camera live stream and uses capsfilter to create a branch for preview. 2. The data is sent to the Wayland display sink (Waylandsink) for preview.
Encode as MP4 file	1. qtiqmmfsrc captures the camera live stream and uses capsfilter to create a branch for encoding. 2. The stream data is encoded. 3. H264parse is used to parse the captured data. 4. Mp4mux is used to multiplex the data.

2.2.1.12 Wide dynamic range and electronic image stabilization

The **gst-camera-shdr-ldc-eis-example** application shows the staggered high dynamic range (sHDR), lens distortion correction (LDC), and electronic image stabilization (EIS) camera imaging features. For more information, see Overview of camera imaging features.

NOTE This sample application isn't supported on QCS8275 and QCS9075.

The figure shows the pipeline, where the input is received from the camera and the output is displayed using Waylandsink. For more information about the pipeline, see Pipeline flow.



Prerequisites

- If not already done so, Download and install eSDK.
- To access your host device, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is enabled, you can skip this step.

- Push the files from the Linux host computer.

  ```
  scp <filename> root@<IP address of target device>:/etc/
  ```

- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.
- Enable the display:

  ```
  export XDG_RUNTIME_DIR=/dev/socket/weston && export
  WAYLAND_DISPLAY=wayland-1
  ```

- Connect to the speaker to enable the audio. For instructions, see Set up audio hardware.

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

- For sHDR, ensure that the Sensor mode for Digital Overlap mode is enabled and tuned.
- For LDC and EIS, ensure that the sensor and hardware are tuned.

Run the application

The following are the commands to display the output:

- sHDR

  ```
  gst-camera-shdr-ldc-eis-example -s 1 -w 1920 -h 1080
  ```

- LDC

  ```
  gst-camera-shdr-ldc-eis-example -l 1 -w 1920 -h 1080
  ```

- EIS

  ```
  gst-camera-shdr-ldc-eis-example -e 1 -w 1920 -h 1080
  ```

To display the available help options, run the following command in the SSH shell:

```
gst-camera-shdr-ldc-eis-example --help
```

To stop the use case, use **CTRL + C**.

Expected output

- sHDR

Figure 2-60 Output without sHDR

Figure 2-61 Output with sHDR

- LDC

Figure 2-62 Output without LDC

Figure 2-63 Output with LDC

Pipeline flow

The table lists the plugins used in the gst-camera-shdr-ldc-eis-example pipeline:

Plugin	Description
qtiqmmfsrc	Captures the camera feed and sends it to the capsfilter.
Capsfilter	Passes the feed for display.
Waylandsink	Shows the output on the Wayland display.

Overview of camera imaging features

The table provides a brief description of the supported imaging features.

Table 2-33 Camera imaging features

Feature	Description
sHDR	▪ The video high dynamic range feature is supported on camera sensors that operate in the Digital Overlap mode. ▪ The fusion of the low and high exposure frames is enabled in the processor GPU. ▪ Use the sHDR flag to enable the sHDR mode.
LDC	▪ A GPU-based LDC algorithm is supported to rectify barrel distortion artifacts that are inherent with wide field of vision (FoV) lens. ▪ Use the LDC flag to enable the LDC mode.
EIS	▪ A GPU-based EIS algorithm is supported. ▪ Use the EIS flag to enable the EIS mode.

For further information on each feature, see Advanced camera features.

2.2.1.13 Decode JPEG images

The **gst-jpg-decode-example** application allows you to decode JPEG images and view the decoded images on a screen.

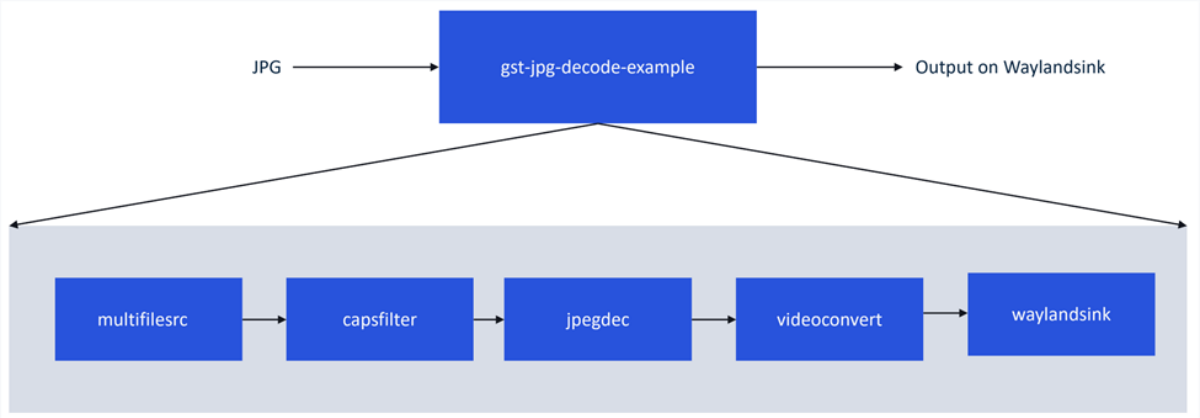


Figure 2-64 gst-jpg-decode-example pipeline

Prerequisites

- If not already done so, Download and install eSDK.
- To access your host device, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is enabled, you can skip this step.
- Push the files from the Linux host computer.

  ```
  scp <filename> root@<IP address of target device>:/etc/
  ```
- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.
- Enable the display:

  ```
  export XDG_RUNTIME_DIR=/dev/socket/weston && export
  WAYLAND_DISPLAY=wayland-1
  ```
- Connect to the speaker to enable the audio. For instructions, see Set up audio hardware.

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

Run the application

Run the use case. Specify the application with parameters such as width and height.

```
gst-jpg-decode-example -w 1920 -h 1080 -i /opt/filename_%d.jpg
```

To display the available help options, run the following command in the SSH shell:

```
gst-camera-switch-example --help
```

To stop the use case, use **CTRL + C**.

Expected output

The decoded image is displayed on the screen for preview.

Pipeline flow

The table lists the plugins used to run the decode JPEG images pipeline:

Plugin	Description
multifilesrc	Reads the video data from sequentially named files.
capsfilter	Enforces constraints on the video data.
jpegdec	Decodes the JPEG video stream.
videoconvert	Converts the video frames from one format to another.
Waylandsink	Displays the video stream on the Wayland display.

2.2.2 Run sample applications for video and audio

The sample applications for the Qualcomm Linux video engine provide custom use cases for video playback, encoding, decoding, transforming, transcoding, and composition. The audio use cases show the custom use cases for playback and recording of the audio files in different formats, in addition to various other capabilities of the Qualcomm Linux audio.

NOTE For QCS8275, audio use cases aren't supported.

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

2.2.2.1 Video wall

The **gst-concurrent-videoplay-composition** application supports concurrent video playback for MP4 AVC (H.264) videos and performs composition on a video wall display.

In the concurrent video playback and composition pipeline, four decode and composition pipelines run concurrently. For more information about the plugins used in this pipeline, see Pipeline flow.

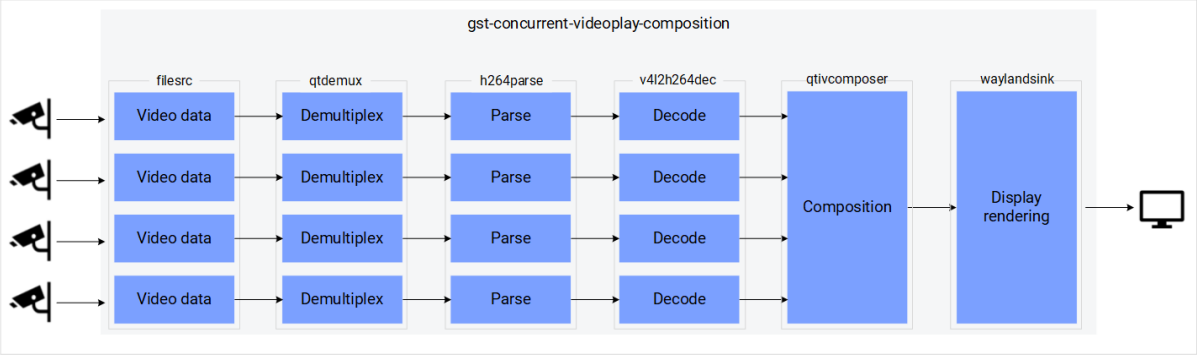
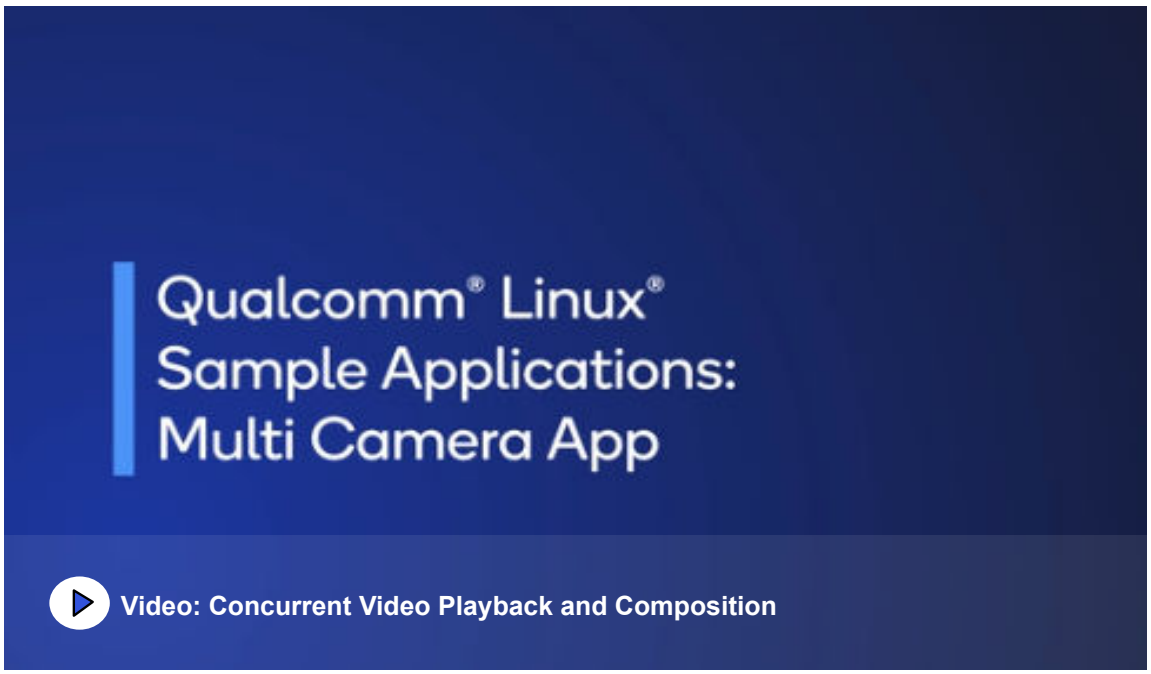


Figure 2-65 gst-concurrent-videoplay-composition pipeline with 4 streams

This video shows you how to stream from several cameras simultaneously, display the feeds side by side, or encode for storage. The video will open in a new tab.



Prerequisites

- If not already done so, Download and install eSDK.
- To access your host device, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is enabled, you can skip this step.
- Push the files from the Linux host computer.

  ```
  scp <filename> root@<IP address of target device>:/etc/
  ```
- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.
- Enable the display:

  ```
  export XDG_RUNTIME_DIR=/dev/socket/weston && export
  WAYLAND_DISPLAY=wayland-1
  ```
- Connect to the speaker to enable the audio. For instructions, see Set up audio hardware.

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

Run the application

The application requires at least one input video file. The input file must be an MP4 AVC (H.264) file.

The table lists the use cases that are supported through the gst-concurrent-videoplay-composition application:

Use case	Description
Video conferencing	■ Simultaneously decodes the video streams of several participants. ■ Useful in a multi-party video conference scenario where the application manages several video streams.
Surveillance systems	Concurrently decodes video feeds from two separate cameras.
Digital signage	Allows simultaneous changes or updates to centrally managed information that's electronically displayed across several locations.

To run the use cases, do the following:

1. Load the AVC-encoded MP4 (H.264) files onto the device.
2. To run the application for your use case, use the following syntax command:

   ```
   gst-concurrent-videoplay-composition -c <number of streams> -i <input-file-1>.mp4 -i <input-file-2>.mp4
   ```

   The table provides a description of the command-line parameters:

Command-line parameter	Description
`-c`	■ Number of streams to decode for composition. ■ Supported values are 2, 4, 8, and 16.
`-i`	Specifies the absolute path to the input video file

   Use the following commands to run the concurrent playback use cases:

   - Concurrent playback of two sessions:

     ```
     gst-concurrent-videoplay-composition -c 2 -i <input-file-1>.mp4 -i <input-file-2>.mp4
     ```

     If one video is provided as input, the same file is played concurrently.
   - Concurrent playback of four sessions:

     ```
     gst-concurrent-videoplay-composition -c 4 -i <input-file-1>.mp4 -i <input-file-2>.mp4 -i <input-file-3>.mp4 -i <input-file-4>.mp4
     ```

To display the available help options, run the following command in the SSH shell:

```
gst-concurrent-videoplay-composition --help
```

To stop the use case, use **CTRL + C**.

Expected output

The individual composed streams are tiled together to display as a unified stream.

Figure 2-66 Expected output for gst-concurrent-videoplay-composition application

Pipeline flow

The table lists the plugins used to run the video wall pipeline:

Plugin	Description
filesrc	Reads the video data.
qtdemux	Demultiplexes the video data.
h264parse	Parses the H.264 video streams.
v4l2h264dec	Decodes the H.264 video streams.
qtivcomposer	Composes the decoded streams and sends them over the display.
Waylandsink	Displays the composed video streams on Wayland display.

Known issues

Video playback is closing or freezing. About an hour later, it continues running with 16 H.264 720p concurrent video playback sessions.

RELATED INFORMATION

"Concurrent video playback (video wall) using Python" on page 180

2.2.2.2 Smart codec

The **gst-smartcodec-example** application reduces bandwidth and storage resources based on scene change estimate thresholds.

The smart codec plugin, qtismartvencbin, uses smart fps, smart I-frame, smart bit rate, and smart regions of interest (ROI) to do the following:

- Decrease the bit rate without compromising the quality.
- Enable efficient use of network and storage resources.

The figure shows the pipeline flow for a two-stream use case. For more information, see Pipeline flow.

Figure 2-67 gst-smartcodec-example pipeline for two streams

Prerequisites

- To access your host device, enable SSH. For instructions, see Log in using SSH.
- Enter the SSH shell and run the use cases:

  ```
  ssh root@<ip-addr of the target device>
  ```

- Push the files from host computer:

  ```
  scp <filename> root@<IP address of target device>:/opt/
  ```

Run the application

Run with a camera source:

```
gst-smartcodec-example -w 1920 -h 1080 -o video.mp4
```

To display the available help options, run the following command in the SSH shell:

```
gst-smartcodec-example --help
```

To stop the use case, use **CTRL + C**.

Expected output

The application generates the output as an encoded MP4 file.

The table lists the file sizes of a 1080p video with and without smart codec.

Table 2-34 Smart codec–file sizes

Smart codec	File size
Enabled	3718 KB
Disabled	17706 KB

Pipeline flow

The table lists the plugins used in the smart codec pipeline:

Plugin	Description
qtiqmmfsrc	1. Captures the video stream. 2. Feeds the stream to capsfilter.
Capsfilter	Branches the source into two separate pipelines. In one pipeline, qtismartvencbin processes the output.
	In the other pipeline, the video undergoes the following: 1. Encoded using v4l2h264enc. 2. Parsed with h264parse. 3. Multiplexed into an MP4 format with mp4mux. 4. Written to a file using filesink.

RELATED INFORMATION

"AI smart codec" on page 85

2.2.2.3 Multi format decode

The **gst-videocodec-concurrent-playback** application supports concurrent video playback for AVC (H.264) and HEVC (H.265) videos. It demonstrates the concurrent decoding capabilities of the Qualcomm Linux video engine, which can process different video codecs simultaneously.

As shown in the figure, the sample application uses H.264 and H.265 videos, both encapsulated in MP4 containers. The pipeline involves several elements for concurrent video playback. For more information about these elements, see Pipeline flow.

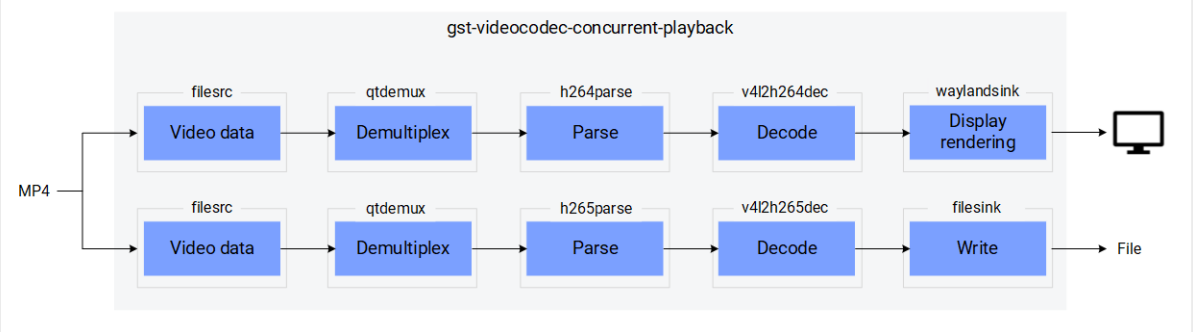


Figure 2-68 gst-videocodec-concurrent-playback pipeline

This video shows how to decode and play several AVC-coded MP4 videos concurrently, and effectively use composition for creating dynamic video wall displays.

Prerequisites

- If not already done so, Download and install eSDK.
- To access your host device, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is enabled, you can skip this step.
- Push the files from the Linux host computer.

  ```
  scp <filename> root@<IP address of target device>:/etc/
  ```
- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.
- Enable the display:

  ```
  export XDG_RUNTIME_DIR=/dev/socket/weston && export
  WAYLAND_DISPLAY=wayland-1
  ```
- Connect to the speaker to enable the audio. For instructions, see Set up audio hardware.

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

Run the application

The table lists the use cases that are supported through the gst-videocodec-concurrent-playback application:

Use case	Description
Video conferencing	▪ Simultaneously decodes the video streams of several participants. ▪ Useful in a multi-party video conference scenario where the application manages several video streams.
Surveillance systems	Concurrently decodes video feeds from two separate cameras, each employing a different encoding method.

To run the use cases, do the following:

1. Load the video files onto the device.
2. Use the following syntax command to run the use case:

```
gst-videocodec-concurrent-playback -i /opt/<h264_file>.mp4 -i /opt/
<h265_file>.mp4 -o /opt/<test>.yuv
```

The table provides a description of the command-line parameters:

Command-line parameter	Description
`-i`	Specifies the absolute path to the H.264 and H.265 input video files. Ensure that the H.264 input file is specified before H.265.
`-o`	Specifies the absolute path to the H.265 output YUV file.

For example, for concurrent playback of multi-format videos, use the following command:

```
gst-videocodec-concurrent-playback  -i /opt/<h264_file>.mp4 -i /opt/
<h265_file>.mp4 -o /opt/<test>.yuv
```

To display the available help options, run the following command in the SSH shell:

```
gst-videocodec-concurrent-playback --help
```

To stop the use case, use **CTRL + C**.

NOTE Due to the large size of YUV data, it's recommended to run this use case for a short duration (less than 5 seconds) to avoid exhausting the storage space.

Expected output

- The H.264 video is displayed to the Wayland display.
- The H.265 HEVC video is written to the YUV file.

  To play the YUV file, do the following:

  a. Pull the YUV dump file from the device.

  b. Use a YUV player running on the host to play the video.

  For example, select the `YUV420planarNV12` format, and use `1920 & 1088` as the stride and scan lines.

  NOTE If a default YUV player isn't available on the host computer, use https://sourceforge.net/projects/raw-yuvplayer/.

Pipeline flow

The table lists the plugins used to run the multi format encode/decode pipeline:

Plugin	Description
filesrc	Reads the video data.
qtdemux	Demultiplexes the video data.
h264parse	Parses the H.264 video stream.
h265parse	Parses the H.265 video stream.

Plugin	Description
v4l2h264dec	Decodes the H.264 video stream using Video4Linux.
v4l2h265dec	Decodes the H.265 video stream using Video4Linux.
Waylandsink	Displays the H.264 video stream on Wayland display.
filesink	Writes the H.265 video to a YUV file for playback.

2.2.2.4 Audio-video playback

The **gst-audio-video-playback** application supports simultaneous audio and video decoding and playback of FLAC/MP3 audio and AVC/HEVC video.

The following are the applicable scenarios to use the gst-audio-video-playback application:

- Media player applications
- Streaming services
- Video conferencing
- Surveillance systems
- Educational platforms
- Entertainment platforms that can play back audio and video files in various formats

As shown in the figure, the pipeline involves several elements for audio and video playback. For more information about these elements, see Pipeline flow.

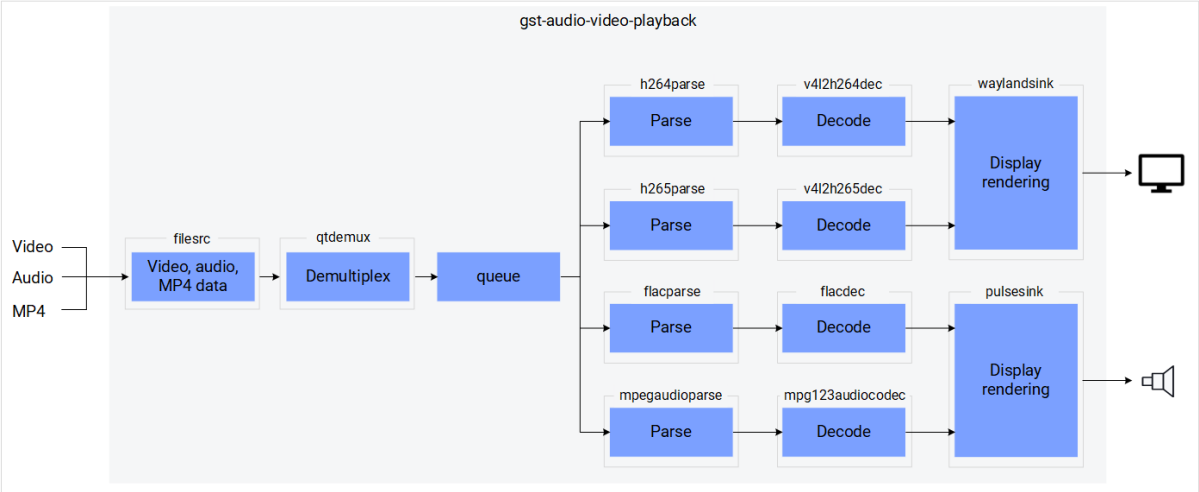


Figure 2-69 gst-audio-video-playback pipeline

Prerequisites

- If not already done so, Download and install eSDK.
- To access your host device, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is enabled, you can skip this step.
- Push the files from the Linux host computer.

```
scp <filename> root@<IP address of target device>:/etc/
```

- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.
- Enable the display:

  ```
  export XDG_RUNTIME_DIR=/dev/socket/weston && export
  WAYLAND_DISPLAY=wayland-1
  ```

- Connect to the speaker to enable the audio. For instructions, see Set up audio hardware.

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

Run the application

1. Load the applicable audio and video files onto the device.
2. Run the use case:

   ```
   gst-audio-video-playback -v 1 -a 1 -i /opt/<filename>.mp4
   ```

   The table provides a description of the command-line parameters:

Command-line parameter	Description
`-v`	Specifies the video codec. ▪ `1` = AVC (H.264) ▪ `2` = HEVC (H.265)
`-a`	Specifies the audio codec. ▪ `1` = FLAC ▪ `2` = MP3
`-i`	Specifies the absolute path to the MP4 input file

   The following list provides commands to run use cases for different audio and video formats:

   - **AVC video with FLAC audio:**

     ```
     gst-audio-video-playback -v 1 -a 1 -i /opt/<filename>.mp4
     ```

   - **AVC video with MP3 audio:**

     ```
     gst-audio-video-playback -v 1 -a 2 -i /opt/<filename>.mp4
     ```

   - **HEVC video with FLAC audio:**

     ```
     gst-audio-video-playback -v 2 -a 1 -i /opt/<filename>.mp4
     ```

   - **HEVC video with MP3 audio:**

     ```
     gst-audio-video-playback -v 2 -a 2 -i /opt/<filename>.mp4
     ```

To display the available help options, run the following command in the SSH shell:

```
gst-audio-video-playback --help
```

To stop the use case, use **CTRL + C**.

Expected output

- The AVC/HEVC video is displayed on the screen.
- The audio plays through the device speaker.

Figure 2-70 Expected output for gst-audio-video-playback application

Pipeline flow

The table lists the plugins used to run the audio-video playback pipeline:

Plugin	Description
filesrc	Reads the video data.
qtdemux	Demultiplexes the video data into separate audio and video streams.
h264parse	Parses the H.264 video stream.
h265parse	Parses the H.265 video stream.
v4l2h264dec	Decodes the H.264 video streams.
v4l2h265dec	Decodes the H.265 video stream.
Waylandsink	Displays the video stream on Wayland display.
flacparse or mpegaudioparse	Parses the audio stream.
flacdec or mpg123audiodec	Decodes the audio stream.
pulsesink	Allows playback of the audio stream.

RELATED INFORMATION

"Audio-video use cases" on page 318

2.2.2.5 Video transcoding

The **gst-video-transcode-example** application supports video transcoding of AVC-to-HEVC or HEVC-to-AVC format. A few use cases can be video editing, security camera, and augmented reality.

The figure shows the pipeline, which reads the incoming stream, decodes and encodes it, and either displays the transcoded output on a screen, or saves it to a file. For information about the plugins used in this pipeline, see Pipeline flow.

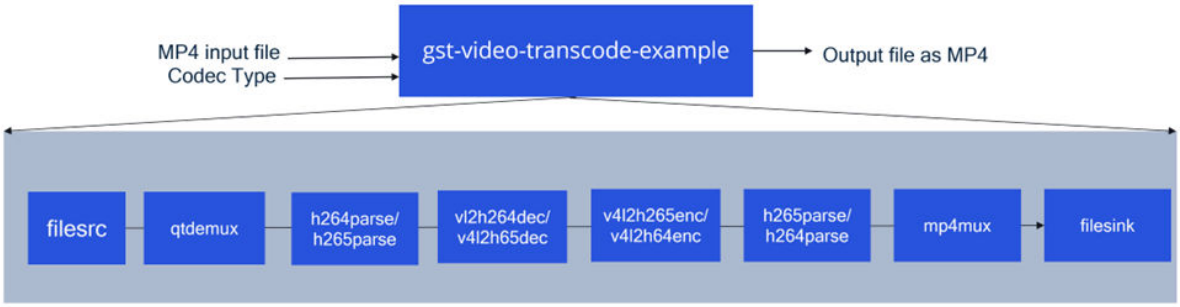


Figure 2-71 gst-video-transcode-example pipeline

Prerequisites

- If not already done so, Download and install eSDK.
- To access your host device, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is enabled, you can skip this step.
- Push the files from the Linux host computer.

  ```
  scp <filename> root@<IP address of target device>:/etc/
  ```
- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.
- Enable the display:

  ```
  export XDG_RUNTIME_DIR=/dev/socket/weston && export
  WAYLAND_DISPLAY=wayland-1
  ```
- Connect to the speaker to enable the audio. For instructions, see Set up audio hardware.

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

Run the application

The table lists the actions that can be performed for the common use cases of gst-video-transcode-example application:

Use case	Actions
Video editing	Rotate, flip, or resize videos.
Security camera	Invert or rotate the video image to compensate for situations where the camera is improperly mounted.
Augmented reality	Scale or rotate the video feed to synchronize with the virtual components.

Do the following to run the transcoding use cases:

1. Load the video files onto the device.
2. Transcode the video file:

   ```
   gst-video-transcode-example -i /opt/<input-file>.mp4 -c <format> -o /opt/
   <output-file>.mp4
   ```

   The table provides a description of the command-line parameters:

Command-line parameter	Description
`-i`	Specifies the absolute path to the MP4 input file.
`-c`	Specifies the video codec of the *input* file. ▪ `1` = AVC (H.264) ▪ `2` = HEVC (H.265)
`-o`	Specifies the absolute path to where the transcoded MP4 file is saved.

   Run the use cases for different video formats:

   - Transcode from AVC to HEVC:

     ```
     gst-video-transcode-example -i /opt/<avc>.mp4 -c 1 -o /opt/<hevc>.mp4
     ```

   - Transcode from HEVC to AVC:

     ```
     gst-video-transcode-example -i /opt/<hevc>.mp4 -c 2 -o /opt/<avc>.mp4
     ```

To display the available help options, run the following command in the SSH shell:

```
gst-video-transcode-example --help
```

To stop the use case, use **CTRL + C**.

Expected output

The video stream is transcoded from H.264 to H.265 and back again to the H.264 format.

Pipeline flow

The table lists the plugins used in the video transcoding pipeline:

Plugin	Description
filesrc	Reads the video streams.
qtdemux	Demultiplexes the streams into separate audio and video streams.
h264parse	Parses the H.264 video stream.
h265parse	Parses the H.265 video stream.
v4l2h264dec	Decodes the H.264 video stream using Video4Linux.
v4l2h265dec	Decodes the H.265 video stream using Video4Linux.
Waylandsink	Displays the H.264 video stream on Wayland display.
filesink	Writes the H.265 video to a YUV file for playback.

The table provides information on how the decoded video streams are handled to generate the transcoded output:

Video format	Description
H.264	1. v4l2h265enc encodes the decoded H.264 video into H.265 format. 2. h265parse parses the H.265 video stream. 3. mp4mux multiplexes the stream into an MP4 container. 4. filesink writes the stream data to a file. The AVC (H.264) video is transcoded to HEVC (H.265) format.
H.265	1. v4l2h264enc encodes the decoded H.265 video into H.264 format. 2. h264parse parses the H.264 video stream. 3. mp4mux multiplexes the stream into an MP4 container. 4. filesink writes the stream data to a file. The HEVC (H.265) video is transcoded to AVC (H.264) format.

RELATED INFORMATION

"Transform and Transcode use cases" on page 303

2.2.2.6 Video playback

The **gst-video-playback-example** application allows you to decode and playback a video file. The standard playback options such as play, pause, resume, and fast forward are supported.

NOTE This sample application isn't supported on QCS8275 and QCS9075.

The figure shows the pipeline workflow of the video playback GStreamer application. The file is read, demultiplexed, parsed, decoded, and displayed on a screen. For more information about the plugins, see Pipeline flow.

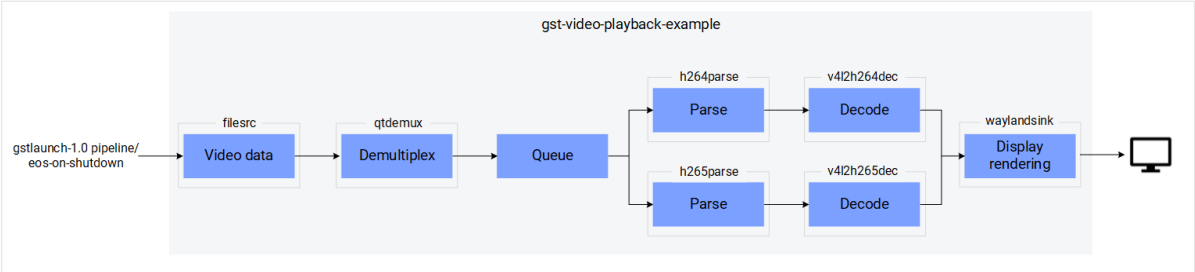


Figure 2-72 gst-video-playback-example pipeline

Prerequisites

- If not already done so, Download and install eSDK.
- To access your host device, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is enabled, you can skip this step.
- Push the files from the Linux host computer.

  ```
  scp <filename> root@<IP address of target device>:/etc/
  ```
- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.

- Enable the display:

  ```
  export XDG_RUNTIME_DIR=/dev/socket/weston && export
  WAYLAND_DISPLAY=wayland-1
  ```

- Connect to the speaker to enable the audio. For instructions, see Set up audio hardware.

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

To use this application effectively, you should be aware of the GStreamer pipeline construction.

Run the application

1. Load the video file onto the device.
2. Play back the video file:

   ```
   gst-video-playback-example -e filesrc location=<input-file>.mp4 !
   qtdemux ! queue ! h264parse ! v4l2h264dec capture-io-mode=4 output-io-
   mode=4 ! video/x-raw,format=NV12 ! waylandsink enable-last-sample=false
   fullscreen=true
   ```

   The table provides a description of the command-line parameters:

Command-line parameter	Description
`<input-file>`	Specifies the MP4 input file to playback
`<video-parser>`	Specifies the video parser. ▪ `h264parse` for H.264 ▪ `h265parse` for H.265
`<video-decoder>`	Specifies the video decoder. ▪ `v4l2h264dec` for H.264 ▪ `v4l2h265dec` for H.265

   Run the video playback use cases:

   - Playback AVC (H.264) video:

     ```
     gst-video-playback-example -e filesrc location=<avc_file>.mp4 !
     qtdemux ! queue ! h264parse ! v4l2h264dec capture-io-mode=4 output-io-
     mode=4 ! waylandsink enable-last-sample=false fullscreen=true
     ```

   - Playback HEVC (H.265) video:

     ```
     gst-video-playback-example -e filesrc location=<hevc_file>.mp4 !
     qtdemux ! queue ! h265parse ! v4l2h265dec capture-io-mode=4 output-io-
     mode=4 ! waylandsink enable-last-sample=false fullscreen=true
     ```

To display the available help options, run the following command in the SSH shell:

```
gst-video-playback-example --help
```

To stop the use case, use **CTRL + C**.

Expected output

The video is captured, decoded, and played on the local display.

Figure 2-73 Expected output for gst-video-playback-example application

Pipeline flow

The table lists the plugins used to run the video playback pipeline:

Plugin	Description
filesrc	Reads the AVC or HEVC MP4 file.
qtdemux	Demultiplexes the AVC or HEVC MP4 file.
h264parse	Parses the video stream for H.264 format.
h265parse	Parses the video stream for H.265 format.
v4l2h264dec	Decodes the video stream for H.264 format.
v4l2h265dec	Decodes the video stream for H.265 format.
Waylandsink	Displays the video in full-screen mode on a Wayland display.

Known issues

The rewind playback option isn't supported

RELATED INFORMATION

"Video transformation using OpenCV" on page 178
"Video playback use cases" on page 313

2.2.2.7 Audio and video encode

The **gst-audio-video-encode** application encodes audio and video data, which is stored in a user-specified output file. A few use cases for this application are streaming services, video conferencing, and surveillance systems.

The figure shows the pipeline, which receives the video and audio feed, encodes, and parses the video stream. It then captures the audio stream and encodes it in MP3 format. Both streams are multiplexed and written to a file.

For information about the plugins used in the pipeline, see Pipeline flow.

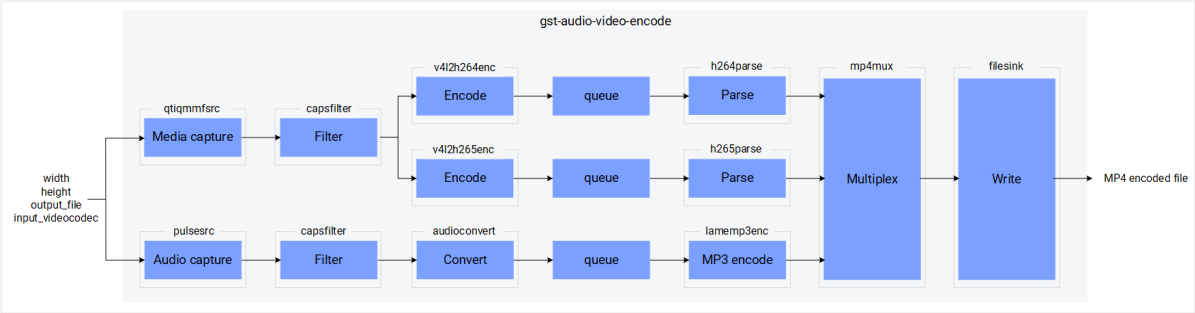


Figure 2-74 gst-audio-video-encode pipeline

Prerequisites

- If not already done so, Download and install eSDK.
- To access your host device, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is enabled, you can skip this step.
- Push the files from the Linux host computer.

  ```
  scp <filename> root@<IP address of target device>:/etc/
  ```
- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.
- Enable the display:

  ```
  export XDG_RUNTIME_DIR=/dev/socket/weston && export
  WAYLAND_DISPLAY=wayland-1
  ```
- Connect to the speaker to enable the audio. For instructions, see Set up audio hardware.

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

Run the application

The table lists the actions that can be performed for the common use cases of gst-audio-video-encode application:

Use case	Actions
Streaming services	Encode audio and video streams for live internet streaming.
Security camera	Invert or rotate the video image to compensate for situations where the camera is improperly mounted.
Augmented reality	Scale or rotate the video feed to synchronize with the virtual components.

Use the following syntax command to run the application:

```
gst-audio-video-encode -w <width> -h <height> -c <input-codec> -o /opt/
<output-file>.mp4
```

The table provides a description of the command-line parameters:

Command-line parameter	Description
`-w`	Specifies the camera width
`-h`	Specifies the camera height
`-c`	Specifies the video codec of the input file: ▪ `1` for AVC (H.264) ▪ `2` for HEVC (H.265)
`output-file`	Provides the absolute path to the output file

Use the following commands to run the video encode use cases:

- AVC:

  ```
  gst-audio-video-encode -w 1920 -h 1080 -c 1 -o /opt/audiovideo.mp4
  ```

- HEVC:

  ```
  gst-audio-video-encode -w 1920 -h 1080 -c 2 -o /opt/audiovideo.mp4
  ```

To display the available help options, run the following command in the SSH shell:

```
gst-audio-video-encode --help
```

To stop the use case, use **CTRL + C**.

Expected output

The encoded audio and video streams are multiplexed to an MP4 container and written to a file.

Pull the files from the host computer:

```
scp root@<IP address of target device>:/opt/<destination directory>
```

Pipeline flow

The table lists the plugins used to run the audio and video stream encoding pipeline:

Plugin	Description
qtiqmmfsrc and capsfilter	Captures the media stream.
v4l2h264enc	Encodes the video for AVC (H.264) format.
v4l2h265enc	Encodes the video for HEVC (H.265) format.
h264parse	Parses the H.264 video stream.
h265parse	Parses the H.265 video stream.
pulsesrc	Captures the audio from the stream, which is filtered using capsfilter.
audioconvert	Converts the audio for encoding.
lamemp3enc	Encodes the audio in MP3 format.
mp4mux	Multiplexes encoded video and audio into an MP4 container.
filesink	Writes the stream into a file.

2.2.2.8 Audio playback

The **gst-audio-decode-example** application demonstrates the playback of audio files in WAV, MP3, and FLAC formats. Separate pipelines for each audio format ensure efficient processing and compatibility.

This sample application provides a basic template that can be extended and adapted.

The figure shows the pipeline, which reads and parses the audio file, and then decodes and plays the audio. For information about the plugins used in the pipeline, see Pipeline flow.

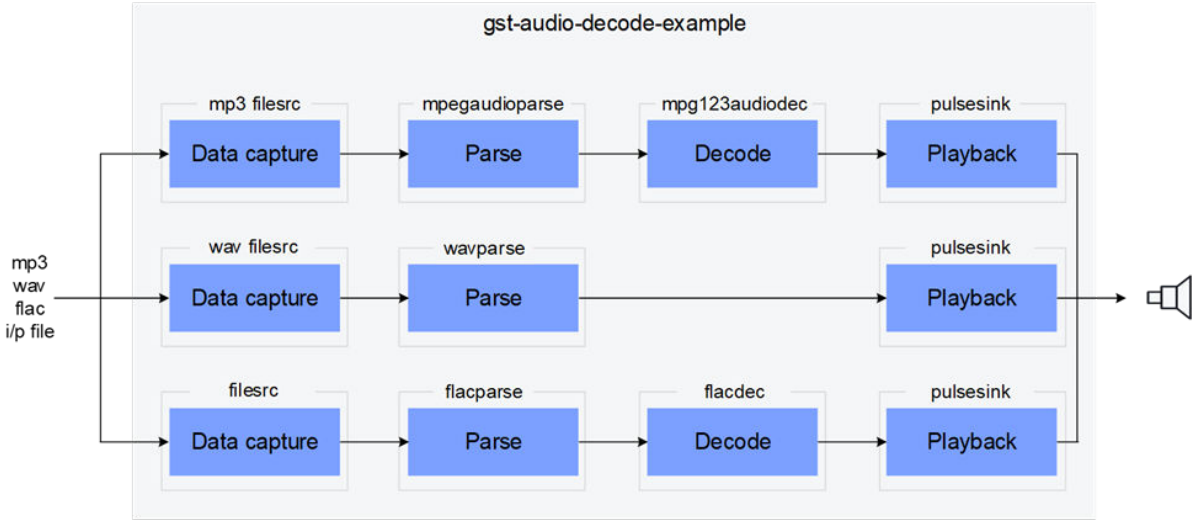


Figure 2-75 gst-audio-decode-example pipeline

Prerequisites

- If not already done so, Download and install eSDK.
- To access your host device, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is enabled, you can skip this step.
- Push the files from the Linux host computer.

  ```
  scp <filename> root@<IP address of target device>:/etc/
  ```
- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.
- Enable the display:

  ```
  export XDG_RUNTIME_DIR=/dev/socket/weston && export
  WAYLAND_DISPLAY=wayland-1
  ```
- Connect to the speaker to enable the audio. For instructions, see Set up audio hardware.

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

Run the application

1. Load the WAV, MP3, or FLAC files onto the device.
2. To run the application for different audio formats, use the following syntax:

   ```
   gst-audio-decode-example -i /opt/<filename> --audio_format=<format>
   ```

   The table provides a description of the command-line parameters:

Command-line parameter	Description
`-i`	Specifies the absolute path to the input audio file to decode
`--audio_format`	Specifies the format of the audio file: ▪ `1` = MP3 ▪ `2` = WAV ▪ `3` = FLAC

   Run the application for MP3, WAV, or FLAC audio formats:

   - Decode MP3:

     ```
     gst-audio-decode-example -i /opt/<filename>.mp3 --audio_format=1
     ```
   - Decode WAV:

     ```
     gst-audio-decode-example -i /opt/<filename>.wav --audio_format=2
     ```
   - Decode FLAC:

     ```
     gst-audio-decode-example -i /opt/<filename>.flac --audio_format=3
     ```

To display the available help options, run the following command in the SSH shell:

```
gst-audio-decode-example --help
```

To stop the use case, use **CTRL + C**.

Expected output

Based on the format (MP3, WAV, FLAC), the decoded audio file is played back.

Pipeline flow

The table lists the plugins used to run the audio playback pipeline:

Plugin	Description
filesrc	Reads the audio files.
▪ mpegaudioparse ▪ wavparse ▪ flacparse	Parses the MPEG, WAV, and FLAC audio files respectively.
▪ mpg123audiodec ▪ flacdec	Decodes the MP3 and FLAC files respectively.
pulsesink	Plays the decoded audio.

RELATED INFORMATION

"Audio use cases" on page 316

2.2.2.9 Audio recording

The **gst-audio-encode-example** application encodes audio in FLAC or WAV formats.

A common use case for this application can be to encode live audio data for streaming or storage.

The figure shows the pipeline, which captures the audio, converts the stream into FLAC or WAV formats, encodes the stream, and stores it in a file. For more information about the plugins used in the pipeline, see Pipeline flow.

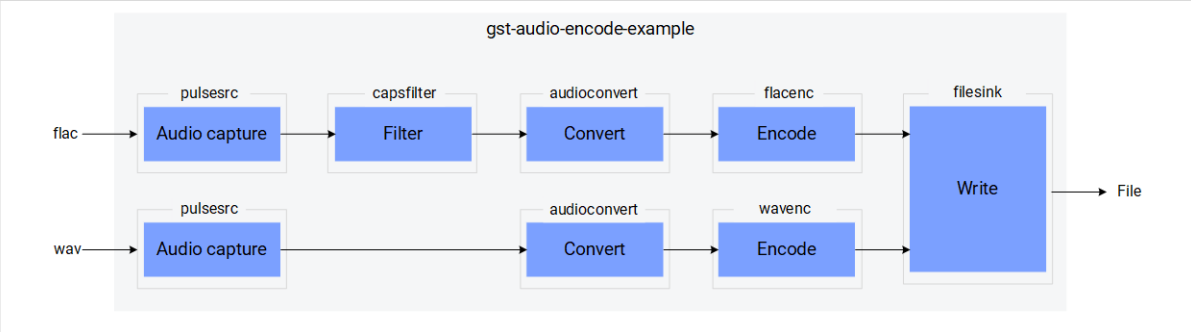


Figure 2-76 gst-audio-encode-example pipeline

Prerequisites

- If not already done so, Download and install eSDK.
- To access your host device, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is enabled, you can skip this step.
- Push the files from the Linux host computer.

```
scp <filename> root@<IP address of target device>:/etc/
```

- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.
- Enable the display:

  ```
  export XDG_RUNTIME_DIR=/dev/socket/weston && export
  WAYLAND_DISPLAY=wayland-1
  ```

- Connect to the speaker to enable the audio. For instructions, see Set up audio hardware.

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

Run the application

For this use case, if you have a live audio source (such as a microphone or an internet radio stream), you can encode this audio data into a more compact or standardized format for efficient transmission or storage.

To run the application for different audio formats, use the following syntax:

```
gst-audio-encode-example -o <path-file> --audio_format=<format>
```

The table provides a description of the command-line parameters:

Command-line parameter	Description
`-o`	Specifies the absolute path to the output audio file.
`-format`	Specifies the format of the audio file: ▪ `1` = FLAC ▪ `2` = WAV

Run the application for FLAC and WAV audio formats:

- Encode FLAC audio:

  ```
  gst-audio-encode-example -o /opt/<filename>.flac --audio_format=1
  ```

- Encode WAV audio:

  ```
  gst-audio-encode-example -o /opt/<filename>.wav --audio_format=2
  ```

To display the available help options, run the following command in the SSH shell:

```
gst-audio-encode-example --help
```

To stop the use case, use **CTRL + C**.

Expected output

The captured audio is encoded and written to a file based on the chosen format (FLAC or WAV).

Pull the files from the host computer:

```
scp root@<IP address of target device>:/opt/<destination directory>
```

Pipeline flow

In the audio encoding, the pipeline differs slightly based on the encoding format.

The table lists the plugins used to run the audio recording pipeline:

Plugin	FLAC	WAV
pulsesrc	Captures audio from the system sound device.	
capsfilter	Ensures that the audio is in a format suitable for the FLAC encoder.	–
audioconvert	Converts the audio to the required format.	
▪ flacenc ▪ wavenc	flacenc encodes the audio into FLAC format.	wavenc encodes the audio into WAV format.
filesink	Writes the encoded audio to a file.	

RELATED INFORMATION

"Audio use cases" on page 316

2.2.2.10 Video transformation

The **gst-transform-example** application supports rotation, flipping, and scaling operations.

You can use this application for a security camera. Adjust the orientation of the displayed image using the flip and mirror operations on the CCTV camera. For more information about the transformation properties, see qtivtransform.

The figure shows the pipeline, which takes the captured video stream, transforms it, and either displays it or saves it to a file. For information about the plugins used in the pipeline, see Pipeline flow.

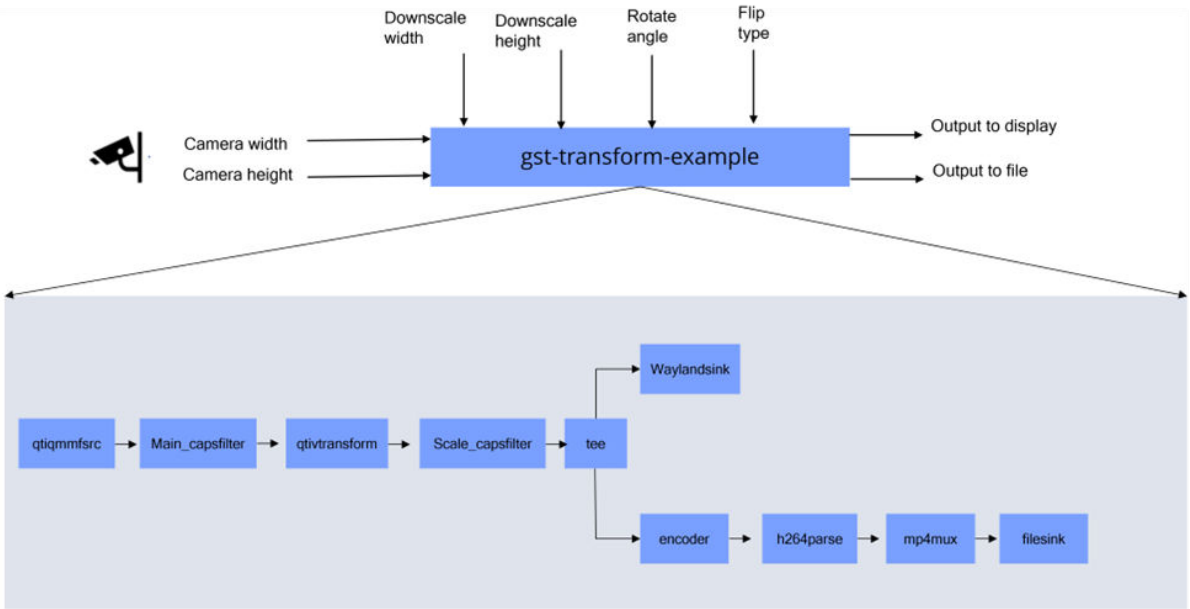


Figure 2-77 gst-transform-example pipeline

Prerequisites

- If not already done so, Download and install eSDK.
- To access your host device, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is enabled, you can skip this step.
- Push the files from the Linux host computer.

  ```
  scp <filename> root@<IP address of target device>:/etc/
  ```
- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.
- Enable the display:

  ```
  export XDG_RUNTIME_DIR=/dev/socket/weston && export
  WAYLAND_DISPLAY=wayland-1
  ```
- Connect to the speaker to enable the audio. For instructions, see Set up audio hardware.

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

Run the application

1. Rotate, downscale, or flip the video:
   - Use camera source and save output to a file.
     - Perform operations such as rotate, flip, upscale/downscale simultaneously:

       ```
       gst-transform-example -r 90 -f 2 --output_width 1920 --output_height
       1080  -o /opt/video_transform.mp4 -i <inputfile>.mp4
       ```
     - Rotate:

       ```
       gst-transform-example -r 90 --input_width 3840 --input_height 2160 -
       o <output_file>.mp4
       ```
     - Downscale/upscale:

       ```
       gst-transform-example --input_width 3840 --input_height 2160 --
       output_width 1920 --output_height 1080 -o <output_file>.mp4
       ```
     - Flip:

       ```
       gst-transform-example -f 2 --input_width 3840 --input_height 2160 -o
       <output_file>.mp4
       ```
   - Use file source and save output to a file.
     - Perform operations such as rotate, flip, upscale/downscale simultaneously:

       ```
       gst-transform-example -r 90 -f 2 -i <inputfileh264>.mp4 --
       output_width 1920 --output_height 1080 -o <output_file>.mp4
       ```
     - Rotate:

       ```
       gst-transform-example -r 90 -i <inputfileh264>.mp4 -o
       <output_file>.mp4
       ```

- Downscale/upscale:

  ```
  gst-transform-example -i <inputfileh264>.mp4 --output_width 1920 --
  output_height 1080 -o <output_file>.mp4
  ```

- Flip:

  ```
  gst-transform-example -f 2 -i <inputfileh264>.mp4 -o
  <output_file>.mp4
  ```

To display the available help options, run the following command in the SSH shell:

```
gst-transform-example --help
```

To stop the use case, use **CTRL + C**.

The table provides a description of the command-line parameters:

Command-line parameter	Description
`-h, --help`	Help
`--output_width`	Downscale/upscale width. ■ The default value is 1920. ■ You can scale from 1080 to 720 or from 720 to 1080.
`--output_height`	Downscale/upscale height. The default value is 1080.
`-o`	Absolute path to generate the file.
`-r`	Image rotate. The values are 0, 90, 180, 270.
`--input_width`	Camera width. The default value is 1920.
`--input_height`	Camera height. The default value is 1080.
`-f`	Flip the image. The values are: ■ `0`: Don't flip (Default) ■ `1`: Flip horizontally ■ `2`: Flip vertically ■ `3`: Flip both horizontally and vertically
`-I`, `--input_file`	Absolute path of the input file. For example,`-I /opt/inputfileh264.mp4`
`-o`,`--output_file`	Absolute path to generate the output file. For example,`-o /opt/video_transform.mp4`

Expected output

This pipeline supports simultaneous live preview and recording of the transformed video stream.

It provides two outputs:

- A video recording that's stored as an MP4 file in the specified path.
- A preview on a display device.

Pull the files from the host computer:

```
scp root@<IP address of target device>:/opt/<destination directory>
```

Figure 2-78 Expected output for gst-transform-example application–Preview

Pipeline flow

The table lists the plugins used to run the video transforming pipeline:

Plugin	Description
qtiqmmfsrc and filesrc	Captures the video data.
capsfilter	Enforces constraints on the raw video data.
qtivtransform	Transforms the video data. After transformation, capsfilter is applied again.
tee	Splits the transformed video data in two output paths: ■ Live preview ■ Save to file
Waylandsink	Displays the video in live preview.
h264parse	Parses the encoded H.264 video stream.
mp4mux	Multiplexes the stream into an MP4 container.
filesink	Writes the video data to a file.

RELATED INFORMATION

"Transform and Transcode use cases" on page 303

2.2.2.11 Video composition

The **gst-weston-composition-example** application allows you to arrange live camera input or offline files on the screen using the picture-in-picture (PiP) or side by side display formats. A few use cases of this application are video conferencing and online educational courses.

The `gst-weston-composition-example` application involves several pipelines and elements. For more information, see Pipeline flow.

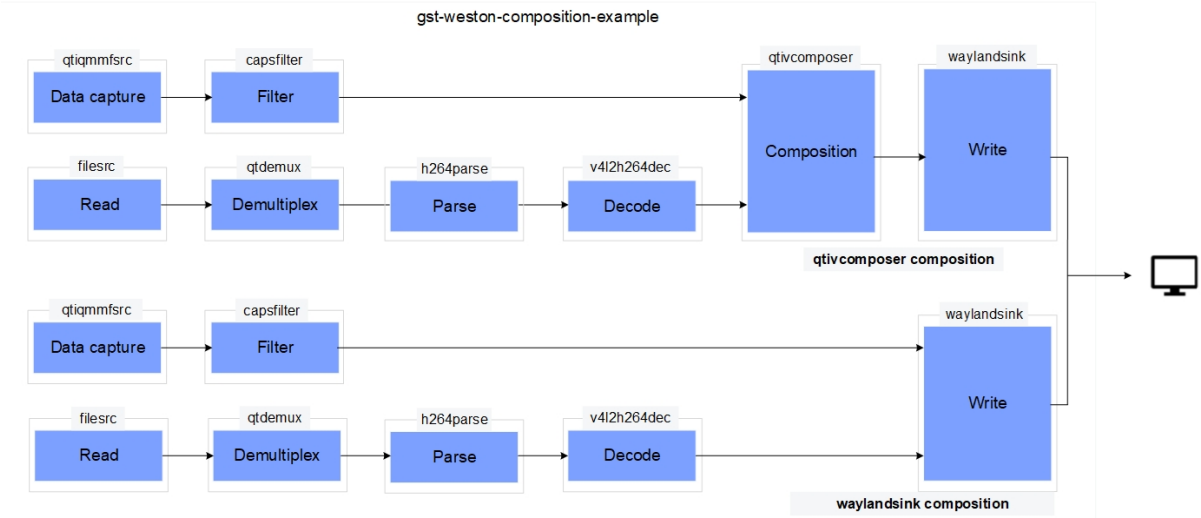


Figure 2-79 gst-weston-composition-example pipeline

Prerequisites

- If not already done so, Download and install eSDK.
- To access your host device, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is enabled, you can skip this step.
- Push the files from the Linux host computer.

  ```
  scp <filename> root@<IP address of target device>:/etc/
  ```
- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.
- Enable the display:

  ```
  export XDG_RUNTIME_DIR=/dev/socket/weston && export
  WAYLAND_DISPLAY=wayland-1
  ```
- Connect to the speaker to enable the audio. For instructions, see Set up audio hardware.

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

Run the application

The table lists the use cases that supported through the gst-weston-composition-example application:

Use case	Description
Video conferencing	■ Simultaneously displays several video streams from many participants. ■ Videos are displayed in either PiP or side by side format. ■ Useful in a multi-party video conference scenario where the application manages several video streams.
Online education	Enhances the learning experience by enabling the PiP or side by side display of the instructional video

For PiP and side by side display formats, you can use qtivcomposer plugin. Do the following to run the use case with either of the plugins:

1. Load the MP4 file onto the device.
2. Run the application:
   - Compose PiP:

     ```
     gst-weston-composition-example -t 0 -i /opt/<h264_file>.mp4
     ```

     Compose side by side:

     ```
     gst-weston-composition-example -t 1 -i /opt/<h264_file>.mp4
     ```

To display the available help options, run the following command in the SSH shell:

```
gst-weston-composition-example --help
```

To stop the use case, use **CTRL + C**.

The table provides a description of the command-line parameters:

Command-line parameter	Description
`-t`	Specifies the type of composition: ■ `0` = Picture-in-picture ■ `1` = Side by side
`-i`	Specifies the absolute path to the input video file.

Expected output

The video is rendered as PiP and side by side display on the screen.

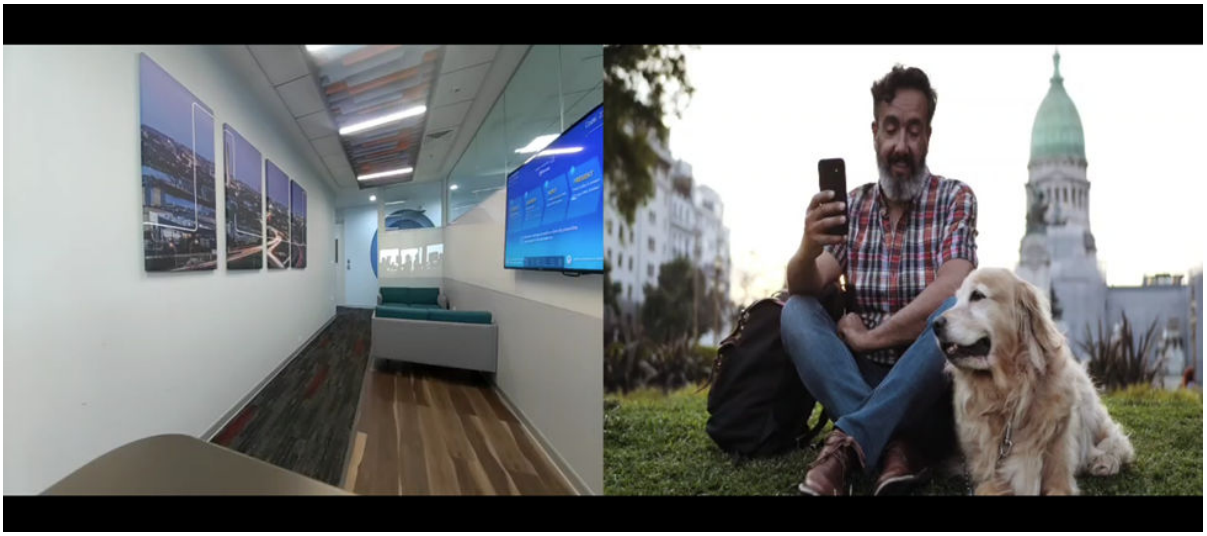


Figure 2-80 Expected output for gst-weston-composition-example application

Pipeline flow

The table lists the plugins used to run the video composition pipeline:

Plugin	Description
qtiqmmfsrc	Captures the video from a source.
capsfilter	Filters the video file.
filesrc	Reads the video data.
qtdemux	Demultiplexes the video data.
h264parse	Parses the H.264 video.
v4l2h264dec	Decodes H.264 video from a file.
qtivcomposer	Composes the decoded video.
Waylandsink	Composes and renders the video on the Wayland display.

Video composition use cases

3 Python applications

Qualcomm Linux integrates support for Python through the gstreamer1.0-python package. Qualcomm IM SDK supports Python bindings that allow you to create GStreamer-based applications in Python. The Python bindings for all the GST plugins are enabled through the `gst-python` utility.

Run the custom use cases using Python to use the multimedia and AI/ML features of Qualcomm Linux.

NOTE As a reference, this guide provides the following Python-based sample applications with Python bindings. Complete the Prerequisites before running these applications.

Table 3-1 Python applications

Application	CodeLinaro Git–source path	Demonstrates	QCS6490	QCS8275	QCS9075
Camera encoding	gst-camera-encode.py	Encode a single camera stream and dump the output.	✔	✖	✖
Camera streaming using OpenCV	gst-camera-opencv-resize.py	Uses OpenCV API to apply color conversion or re-size the camera frame.	✔	✔	✔
Video transformation using OpenCV	gst-opencv-transform	Use OpenCV APIs to capture, transform, and play a video stream.	✔	✔	✔
Multi-camera streaming using Python	gst-multi-camera-example	Stream from two camera sensors simultaneously.	✔	✖	✖
Concurrent video playback (video wall) using Python	gst-concurrent-videoplay-composition.py	Supports concurrent video playback and renders them on a screen.	✔	✔	✔
Object detection and display	gst-ai-object-detection.py	Object detection on streams from a camera or a file source and the output is either displayed or saved to a file.	✔	✖	✖
Decode and object detection using RTSP stream	gst-rtspsrc-detection-display.py	Object detection and display with input from an RTSP stream.	✔	✖	✖
Decode JPEG images using Python	gst-jpg-image-decode.py	Decode several JPEG images and preview them on a screen.	✔	✔	✔
Object detection and classification	gst-camera-two-stream-detection-and-classification-side-by-side.py	Object detection and classification of objects in a scene displayed by the AI overlay composer.	✔	✔	✔

Table 3-1 Python applications (cont.)

Application	CodeLinaro Git–source path	Demonstrates	QCS6490	QCS8275	QCS9075
Transform and encode a camera stream	gst-camera-rotate-downscale-file.py	Rotate, downscale, and encode a camera stream.	✔	✔	✔
Camera encode, object detection, and display	gst-camera-two-stream-encode-file-detection-display.py	Encode a camera stream and save it to a file.	✔	✔	✔
Object detection, classification, and segmentation	gst-filesrc-2detection-classification-segmentation-side-by-side.py	Object detection from a scene in a camera stream, classification on scenes from the video stream, and semantic segmentations for the video.	✔	✔	✔
Parallel inference using Python	gst-parallel-inference.py	Several AI/ML models run in parallel (object detection, classification, segmentation, and pose detection) on streams from a camera, file source, or RTSP.	✔	✖	✔
Daisy chain detection and pose detection using Python	gst-ai-daisychain-detection-pose.py	Perform cascaded object detection and pose detection with a camera source, file source, or an RTSP stream.	✔	✔	✔

3.1 Prerequisites

Complete these preconditions before running the Python-based sample applications.

- To create the directory on the device, run the following commands in a shell:

  ```
  mount -o remount,rw /
  mkdir -p /opt/data/
  ```

- Ensure that the model and label files are available on the device. For instructions, see Download model and label files.
- Rename the model and label files according to the application in use.
- Enable the display:

  ```
  export XDG_RUNTIME_DIR=/dev/socket/weston && export
  WAYLAND_DISPLAY=wayland-1
  ```

- Push the files from the host computer to the target device:

  ```
  scp <video.mp4> root@<IP address of target device>:/opt/data/
  ```

- To enable camera, run `setenforce 0` in the target device shell.

If you face issues while enabling camera or display, see troubleshooting.

See also

- Camera Guide
- Video Guide
- Audio Guide

3.2 Camera encoding

The **gst-camera-encode-example** application allows you to record and encode a single camera stream.

NOTE This sample application isn't supported on QCS8275 and QCS9075.

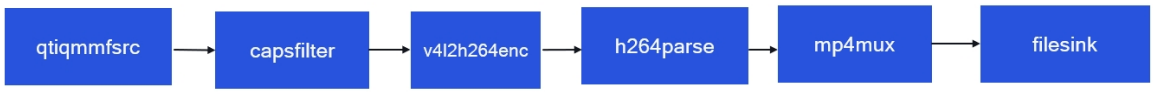


Figure 3-1 Pipeline for camera encoding

For information about the plugins used in this pipeline, see Pipeline flow.

Run the application

1. Ensure that you complete the Prerequisites.
2. Run the camera encode script on the target device:

   ```
   gst-camera-encode.py
   ```

To display the available help options, run the following command:

```
gst-camera-encode.py -h
```

Expected output

The output is saved in a file at `/etc/media/recording_720p.mp4`.

Pipeline flow

The table lists the plugins used in the camera encoding pipeline:

Plugin	Description
qtiqmmfsrc	▪ Captures the camera live stream. ▪ Capsfilter is applied to enforce constraints on the raw video data.
v4l2h264enc	Encodes the video using the H.264 format.
H264parse	Parses the video.
Mp4mux	Multiplexes the video into an MP4 container.
Filesink	Writes the video to a file.

3.3 Camera streaming using OpenCV

The **gst-camera-opencv-resize.py** application uses OpenCV APIs to capture a camera frame from a live camera stream, and then apply color conversion or re-size the frame. The re-sized output is displayed on the screen.

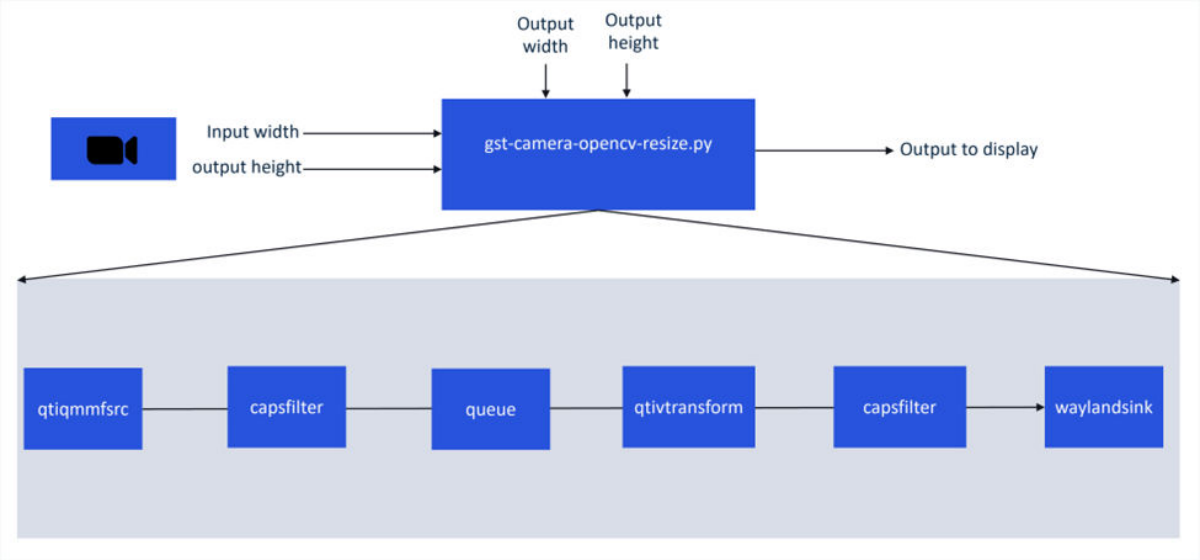


Figure 3-2 gst-camera-opencv-resize.py pipeline

Run the application

1. Ensure that you complete the Prerequisites.
2. Run the application with input from a live camera.

   ```
   gst-camera-opencv-resize.py --inwidth 1280 --inheight 720 --outwidth 640 --
   outheight 480
   ```

   The stream is displayed on the screen.

Expected output

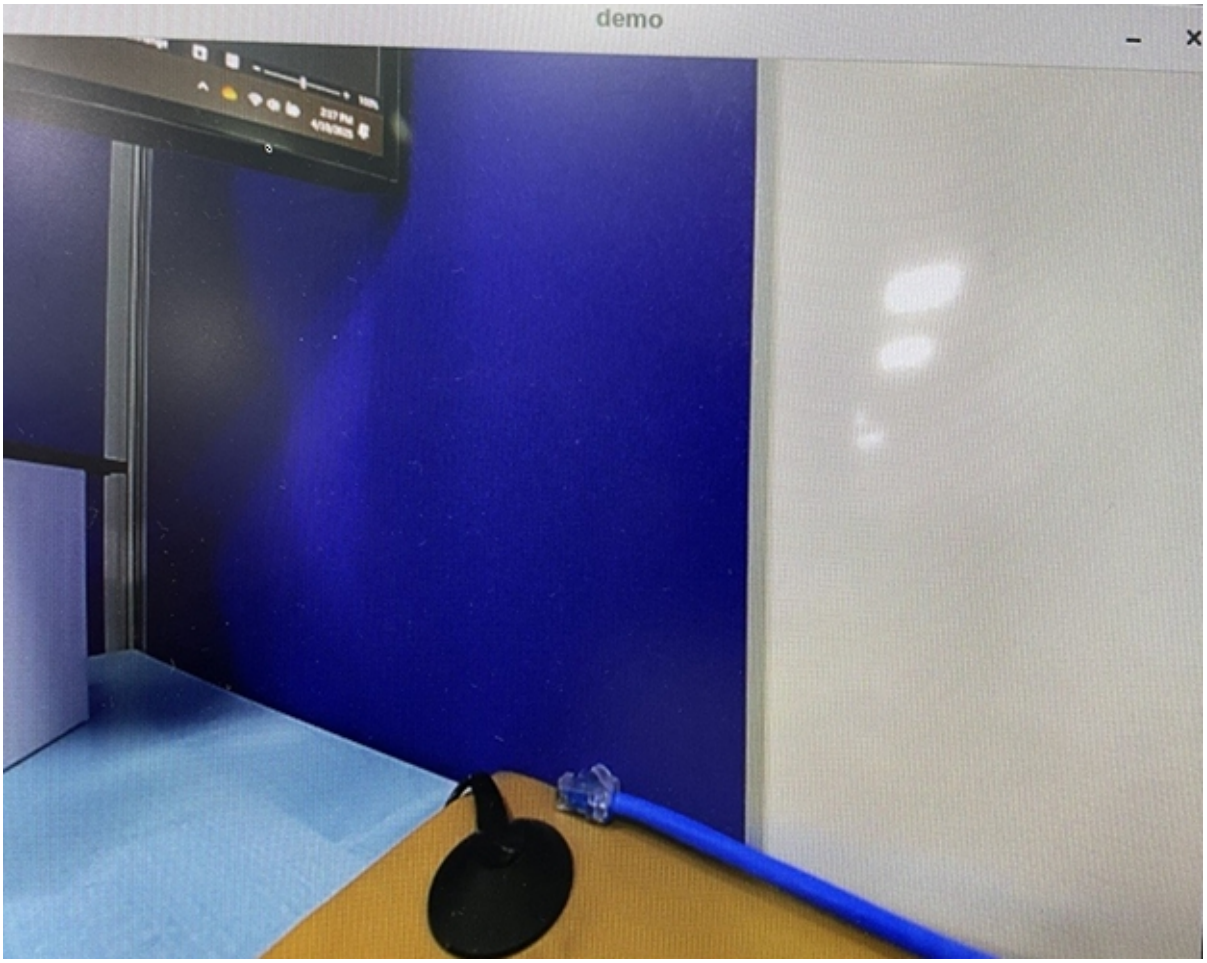


Figure 3-3 Expected output for gst-camera-opencv-resize.py

Pipeline flow

The table lists the plugins used to run the camera streaming with OpenCV pipeline:

Plugin	Description
qtiqmmfsrc	Captures the video stream and feeds it into the capsfilter.
capsfilter	Enforces constraints on the raw video data.
queue	Puts the video data in a queue till the buffer limit is met.
qtivtransform	Transforms the video data. After transformation, capsfilter is applied again.
Waylandsink	Displays the video stream on the Wayland display.

RELATED INFORMATION

"Single camera streaming" on page 110

3.4 Video transformation using OpenCV

The **gst-opencv-transform.py** application captures and transforms a video frame, and then displays the video on the screen.

The figure shows the pipeline, which takes an input video file and parses it. The OpenCV `VideoCapture()` function captures a video frame from the file. The video frame is then converted to a different color format (NV12 to BGR) and displayed on the screen.

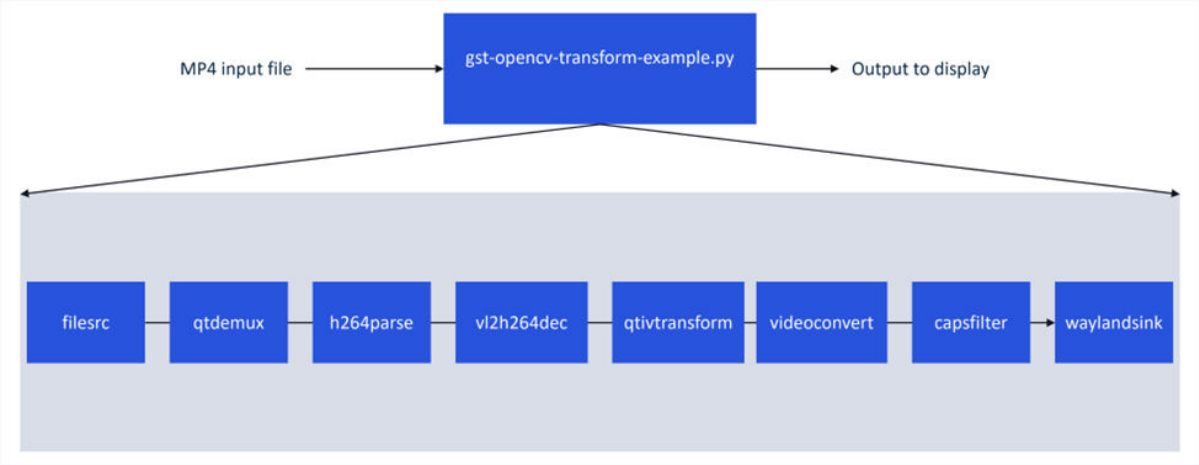


Figure 3-4 gst-opencv-transform.py pipeline

Run the application

1. Ensure that you complete the Prerequisites.
2. Run the application with input from a video file.

   ```
   gst-opencv-transform.py [OPTION?]
   ```

   For example:

   ```
   gst-opencv-transform.py --infile /opt/data/Animals_000_720p_180s_30FPS.mp4
   ```

   The stream is displayed on the screen.

Expected output

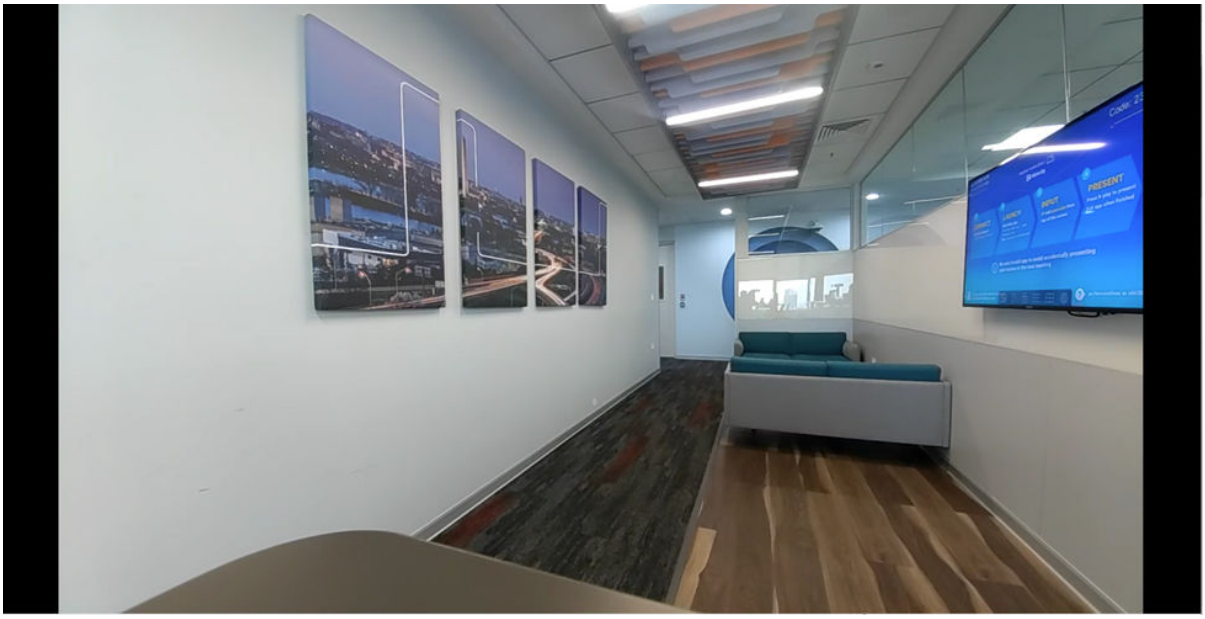


Figure 3-5 Expected output for gst-opencv-transform.py application–Preview

Pipeline flow

The table lists the plugins used to run the video playback with OpenCV pipeline:

Plugin	Description
filesrc	Reads the video data.
qtdemux	Demultiplexes the video data.
h264parse	Parses the H.264 video streams.
v4l2h264dec	Decodes the H.264 video streams.
qtivtransform	Transforms the video data. After transformation, capsfilter is applied again.
videoconvert	Converts the video frames from one format to another.
capsfilter	Enforces constraints on the video data.
Waylandsink	Displays the video stream on the Wayland display.

RELATED INFORMATION

"Video transformation" on page 166
"Transform and Transcode use cases" on page 303
"Video playback" on page 157

3.5 Concurrent video playback (video wall) using Python

The **gst-concurrent-videoplay-composition.py** application supports concurrent video playback for MP4 AVC (H.264) videos and performs composition on a video wall display.

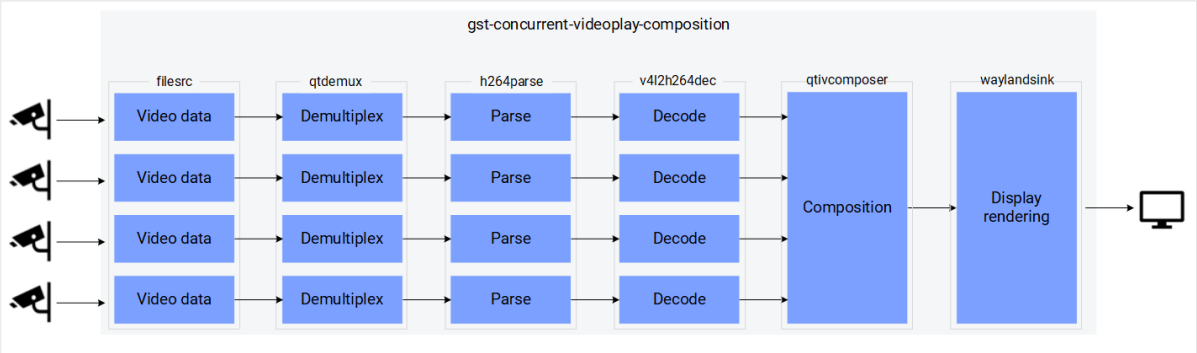


Figure 3-6 gst-concurrent-videoplay-composition.py pipeline with 4 streams

For more information about the plugins used in this pipeline, see Pipeline flow.

Run the application

1. Ensure that you complete the Prerequisites.
2. Run the concurrent video playback composition script on the target device:

   ```
   gst-concurrent-videoplay-composition.py  --infile /opt/data/
   Animals_000_720p_180s_30FPS.mp4 -c 2
   ```

   To change the number of sessions, update `-c` with 2, 4, or 8.

To display the available help options, run the following command:

```
gst-concurrent-videoplay-composition.py -h
```

Expected output

The individual composed streams are tiled together to display as a unified stream.

Figure 3-7 Expected output for gst-concurrent-videoplay-composition.py application

Pipeline flow

The table lists the plugins used to run the video wall pipeline:

Plugin	Description
filesrc	Reads the video data.
qtdemux	Demultiplexes the video data.
h264parse	Parses the H.264 video streams.
v4l2h264dec	Decodes the H.264 video streams.
qtivcomposer	Composes the decoded streams and sends them over the display.
Waylandsink	Displays the composed video streams on Wayland display.

RELATED INFORMATION

"Video wall" on page 144

3.6 Multi-camera streaming using Python

The **gst-multi-camera-example.py** application allows you to stream from two camera sensors simultaneously. The application composes the camera feeds side by side to display on a screen or encodes and stores the video streams to files.

NOTE This sample application isn't supported on QCS8275 and QCS9075.

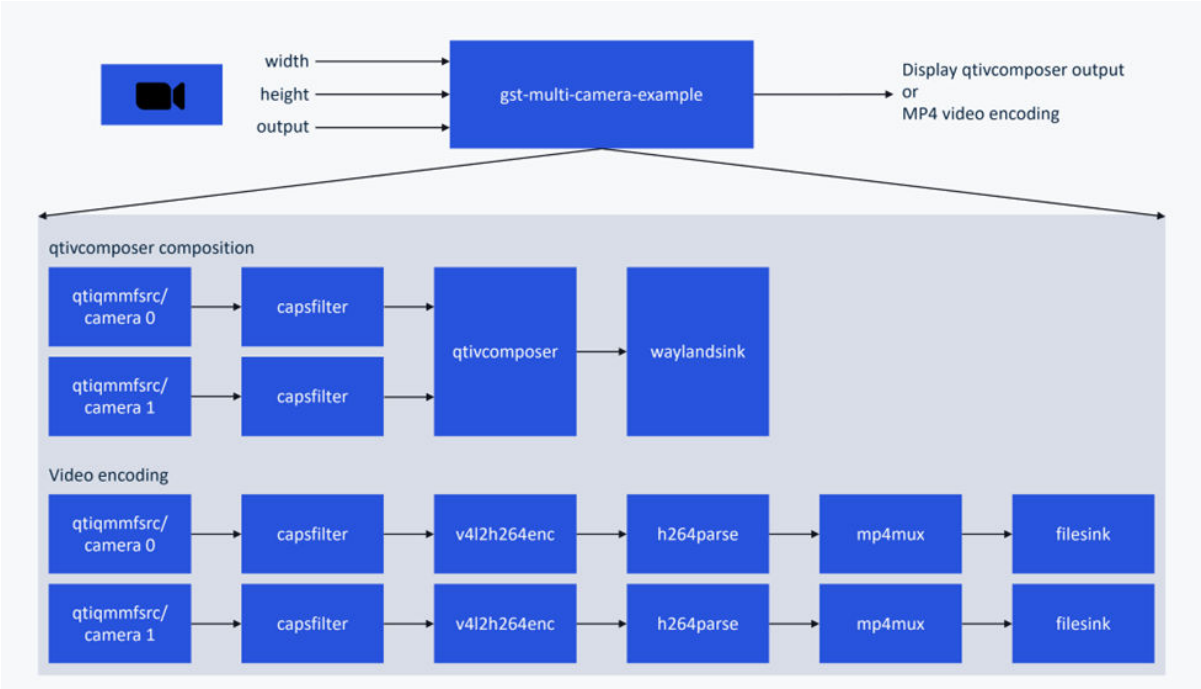


Figure 3-8 gst-multi-camera-example.py pipeline

For information about the plugins used in this pipeline, see Pipeline flow.

Run the application

1. Ensure that you complete the Prerequisites.
2. View the output (preview) on display:

   ```
   gst-multi-camera-stream-example.py -D 0 --width=1920 --height=1080 --
   framerate=30/1
   ```

3. View the encoder output:

   ```
   gst-multi-camera-stream-example.py -D 1 --width=1920 --height=1080
   ```

To display the available help options, run the following command:

```
gst-multi-camera-stream-example.py --help
```

Expected output

For the video composition pipeline, the output is displayed as preview.

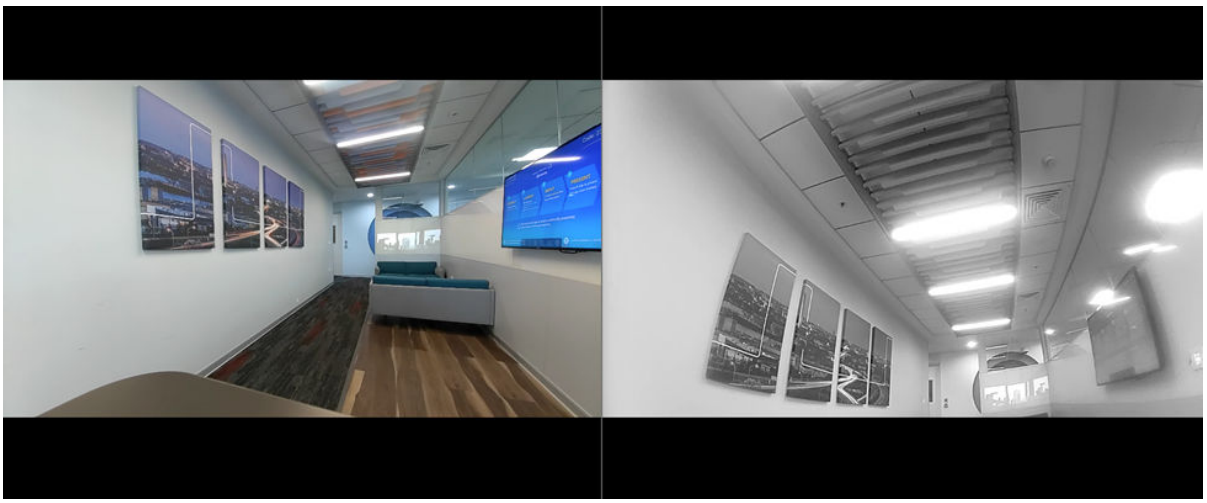


Figure 3-9 Expected output for gst-multi-camera-example.py application–Preview

For video encoding pipeline, the output is saved to a file.

Pipeline flow

The table lists the plugins used in the multi-camera streaming pipeline:

Pipeline	Description
Preview on display	1. qtiqmmfsrc captures video from both camera0 and camera1. 2. Capsfilter is applied to enforce constraints on the raw video data. 3. qtivcomposer composites the video streams and sends the composited video data to Wayland display sink. 4. Waylandsink shows the live preview.
Encoder dump on the device	1. qtiqmmfsrc captures video from both camera0 and camera1. 2. Capsfilter is applied to enforce constraints on the raw video data. 3. v4l2h264enc is used to encode the video using the H.264 format. 4. H264parse is used to parse the video.

Pipeline	Description
	5. Mp4mux is used to multiplex the video into an MP4 container. 6. Filesink is used to write the video to a file.

RELATED INFORMATION

"Multi-camera streaming" on page 115

3.7 Object detection and display

The **gst-ai-object-detection.py** application allows you to detect objects in the camera stream or a file stream and display the results or save the output to a file.

NOTE This sample application isn't supported on QCS8275 and QCS9075.

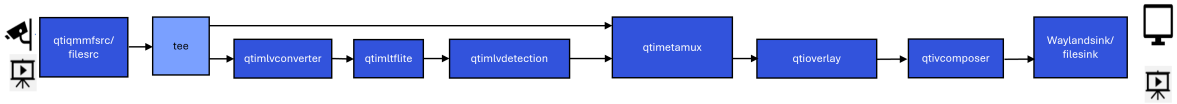


Figure 3-10 Pipeline for object detection and preview

For information about the plugins used in this pipeline, see Pipeline flow.

Run the application

1. Ensure that you complete the Prerequisites.
2. Run various use cases:

   Table 3-2 Sample model and label files for object detection and display python application

Runtime	Models	Labels
LiteRT	*yolov8_det_quantized.tflite*	*yolonas.labels*
Qualcomm Neural Processing SDK	*yolonas.dlc*	*yolonas.labels*

   - Display with the primary and secondary cameras respectively:
     - `gst-ai-object-detection.py -c 0 -f 2 -m /etc/models/yolov8_det_quantized.tflite -l /etc/labels/yolov8.labels -ml "yolov8" -k "YOLOv8,q-offsets=<-107.0, -128.0, 0.0>,q-scales=<3.093529462814331, 0.00390625, 1.0>"`
     - `gst-ai-object-detection.py -c 1 -f 2 -m /etc/models/yolov8_det_quantized.tflite -l /etc/labels/yolov8.labels -ml "yolov8" -k "YOLOv8,q-offsets=<-107.0, -128.0, 0.0>,q-scales=<3.093529462814331, 0.00390625, 1.0>"`
   - Display with input from a video file:

     ```
     gst-ai-object-detection.py -s /path/to/input/video.mp4 -f 2 -m /etc/models/yolov8_det_quantized.tflite -l /etc/labels/yolov8.labels -ml
     ```

```
"yolov8" -k "YOLOv8,q-offsets=<-107.0, -128.0, 0.0>,q-
scales=<3.093529462814331, 0.00390625, 1.0>"
```

- YOLO-NAS with LiteRT:

```
gst-ai-object-detection.py -f 2 -m /etc/models/yolonas.tflite -l /etc/
labels/yolonas.labels -ml "yolo-nas" -k "YOLO-NAS,q-offsets=<37.0, 0.0,
0.0>,q-scales=<3.416602611541748, 0.00390625, 1.0>"
```

- YOLO-NAS with Qualcomm Neural Processing SDK runtime:

```
gst-ai-object-detection.py -f 1 -m /etc/models/yolonas.dlc -l /etc/
labels/yolonas.labels -ml "yolo-nas" --layers="/heads/Mul,/heads/
Sigmoid"
```

- YOLOv8 with LiteRT:

```
gst-ai-object-detection.py -f 2 -m /etc/models/yolov8.tflite -l /etc/
labels/yolov8.labels -ml "yolov8" -k "YOLOv8,q-offsets=<-107.0, -128.0,
0.0>,q-scales=<3.093529462814331, 0.00390625, 1.0>"
```

To display the available help options, run the following command:

```
gst-ai-object-detection.py -h
```

Expected output

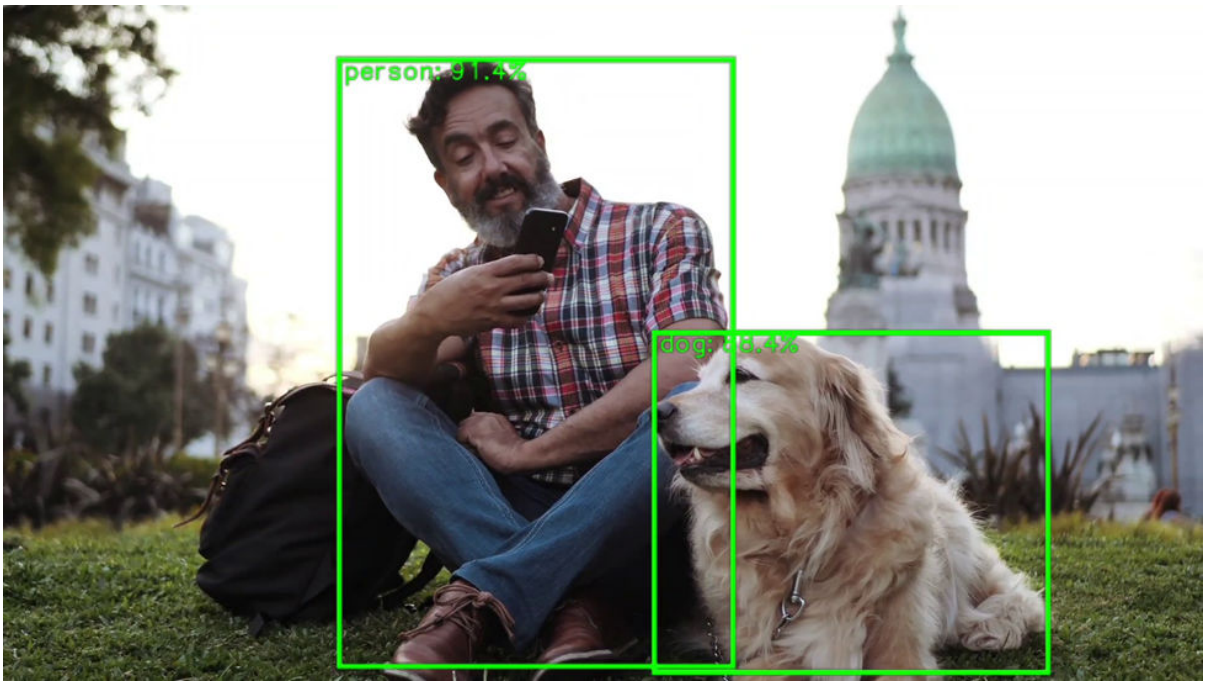


Figure 3-11 Expected output for object detection and display application–Preview

Pipeline flow

Table 3-3 Pipeline processing stages for object detection and preview

Process	Description
qtiqmmfsrc	1. Collects the video stream (source) and creates two copies of the source: ▫ One stream is sent to the qtimetamuxplugin to retain the video stream. ▫ The other stream is sent to an ML inferencing pipeline.
filesrc	Reads the video data.
Preprocessing	
qtimlvconverter	1. Receives the video stream on its sinkpad. 2. Performs preprocessing: ▫ Color conversion ▫ Scaling down/up ▫ Normalization on the stream data when the model expects the floating point values as input 3. Converts the video stream to a tensor stream on its source pad. The object detection model uses this tensor stream for inferencing.
Inferencing	
qtimltflite	1. Loads the object detection model. 2. Modifies the graph for the chosen delegate. 3. Receives the tensor stream on its sinkpad. 4. Runs the inference and produces a tensor stream with the object detection results on its source pad.
Postprocessing	
qtimlvdetection	1. Receives the inference tensors from object detection. 2. Converts the inference tensors on its sinkpad into formats such as video or text that the multimedia plugins can process later. 3. Applies the threshold to the chosen number of results. 4. Loads the corresponding modules for detection models. In this use case, qtimlvdetection does the following: a. Loads the YOLOv8 submodule. b. Produces results as structures of text. c. Sends them to the sinkpad of qtimetamux.
qtimetamux	1. Receives video stream and text stream with bounding box results corresponding to the video stream on its sinkpads. 2. Produces GST buffers with contents of video stream from its sink pad. 3. Adds bounding boxes as GstVideoRegionOfInterest from data sinkpad to GST buffers meta (meta multiplexing) on its source pad.
qtivoverlay	1. Receives the multiplexed stream. 2. Overlays the bounding boxes on the VideoFrame using CL. 3. Produces GST buffers with overlays in its source pad.
Output	

Table 3-3 Pipeline processing stages for object detection and preview (cont.)

Process	Description
Waylandsink	1. Receives the video stream on its sinkpad. 2. Submits the video stream to Weston. 3. Weston renders the video stream and bounding boxes generated for the objects in that scene on a local display device.
Filesink	Writes the video to a file.

RELATED INFORMATION

"Object detection" on page 33

3.8 Decode and object detection using RTSP stream

The **gst-rtspsrc-detection-display.py** script receives an RTSP stream as a source, decodes it, uses YOLOv8 LiteRT model to identify the object in a scene from the camera stream and overlay the bounding boxes over the detected objects. The results are shown on the display.

NOTE This sample application isn't supported on QCS8275 and QCS9075.

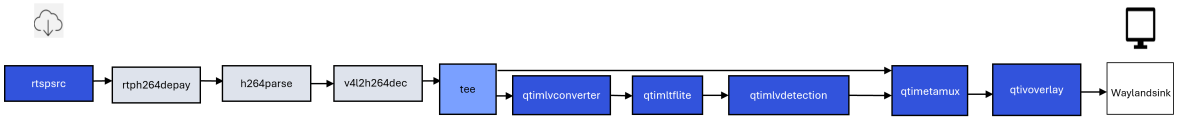


Figure 3-12 Pipeline for decode and object detection using RTSP stream–Preview

Run the application

1. Ensure that you complete the Prerequisites.
2. Run the object detection script on the target device:
   a. Console 1
      – 640 × 480 stream:

      ```
      gst-launch-1.0 -e qtiqmmfsrc camera=0 ! v4l2h264enc capture-io-mode=4 output-io-mode=5 ! h264parse config-interval=1 ! qtirtspbin
      ```

      – 1980 × 1080 stream:

      ```
      gst-launch-1.0 -e qtiqmmfsrc camera=0 ! video/x-raw,format=NV12,width=1920,height=1080,framerate=30/1 ! v4l2h264enc capture-io-mode=4 output-io-mode=5 ! h264parse config-interval=1 ! qtirtspbin
      ```

   b. Console 2:

      ```
      gst-rtspsrc-detection-display.py
      ```

Table 3-4 Default directories for model and label files for object detection using RTSP stream

Runtime	Models	Labels
LiteRT	`/etc/models/YoloV8N_Detection_Quantized.tflite`	`/etc/labels/yolov8n.labels`

To display the available help options, run the following command:

```
gst-rtspsrc-detection-display.py -h
```

Expected output

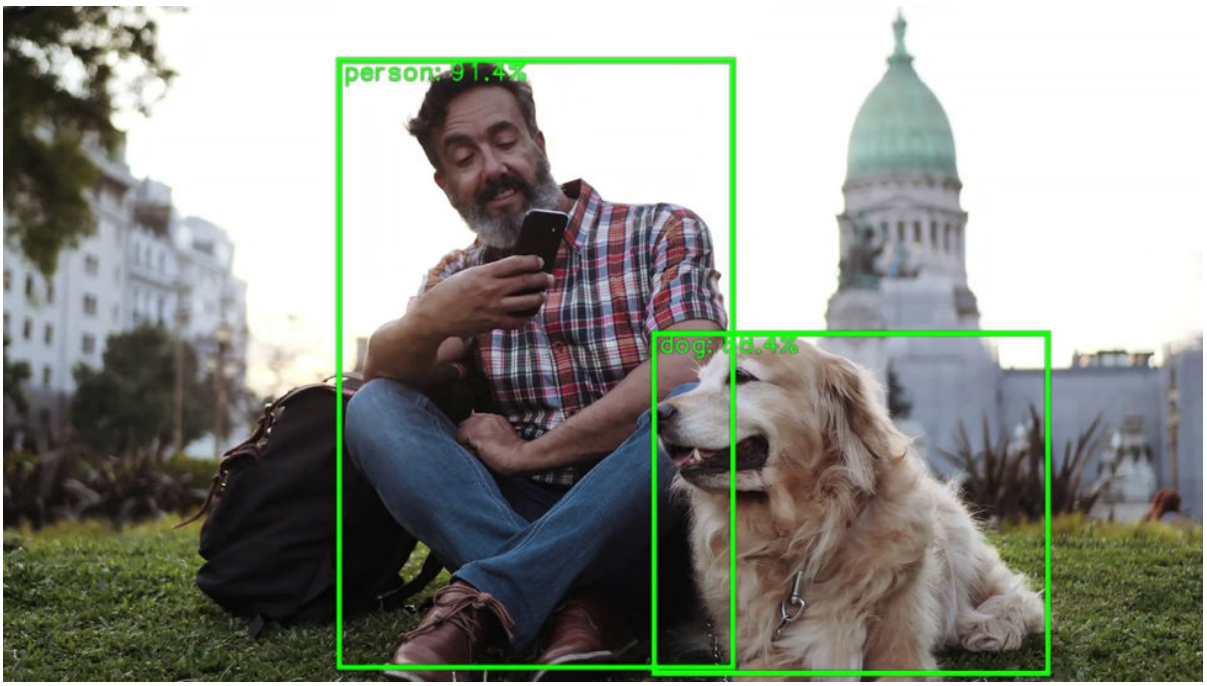


Figure 3-13 Expected output for object detection and display using RTSP stream–Preview

Pipeline flow

The table lists the plugins used in the decode and object detection using RTSP stream pipeline:

Plugin	Description
rtspsrc	Receives the RTSP stream from `rtsp://<ip-address>/live`.
rtph264depay	Extracts the video data from RTSP stream.
h264parse	Parses the H.264 video.
v4l2h264dec	Decodes the video. Then, tee splits the stream for inferencing.

Plugin	Description
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs the following preprocessing on the stream data. This preprocessing is done when the model expects floating-point values as input. a. Color conversion b. Scaling (up or down) c. Normalization The object detection model uses the tensor stream for inferencing.
qtimltflite	1. Loads the model. 2. Modifies the graph for the chosen delegate. 3. Receives the tensor stream on its sinkpad. 4. Runs the inference and produces a tensor stream with the inference results on its source pad.
qtimlvdetection	1. Receives the inference tensors from the object detection model. 2. Converts the inference tensors on its sinkpad into formats like video or text that the multimedia plugins can process later. 3. Applies the threshold to the chosen number of results. 4. Loads the corresponding modules for detection models. In this use case, qtimlvdetection does the following: a. Loads the YOLOv8 submodule. b. Produces results as structures of text. c. Sends them to the sinkpad of qtimetamux.
qtimetamux	1. Receives video stream and text stream with bounding box results corresponding to the video stream on its sinkpads. 2. Produces GST buffers with contents of the video stream from its sink pad. 3. Adds bounding boxes as GstVideoRegionOfInterest from data sinkpad to GST buffers meta (meta muxing) on its source pad.
qtivoverlay	1. Receives the multiplexed stream. 2. Overlays the bounding boxes on the VideoFrame using CL. 3. Produces GST buffers with overlays in its source pad.
Waylandsink	1. Receives the video in its sinkpad 2. Submits the video stream to Weston. 3. Weston renders the video stream on a local display device.

RELATED INFORMATION

"Object detection" on page 33

3.9 Decode JPEG images using Python

The **gst-jpg-image-decode.py** application allows you to decode JPEG images and view the decoded images on a screen.

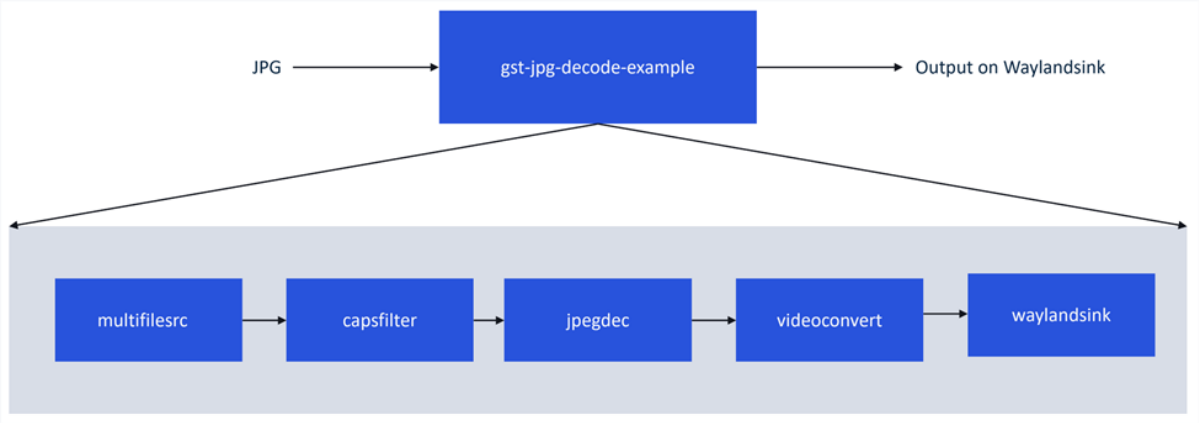


Figure 3-14 gst-jpg-image-decode.py pipeline

Run the application

1. Ensure that you complete the Prerequisites.
2. Run the application on the target device with input from a camera source:

```
gst-jpg-image-decode.py -i /opt/media/frame%d.JPG
```

Expected output

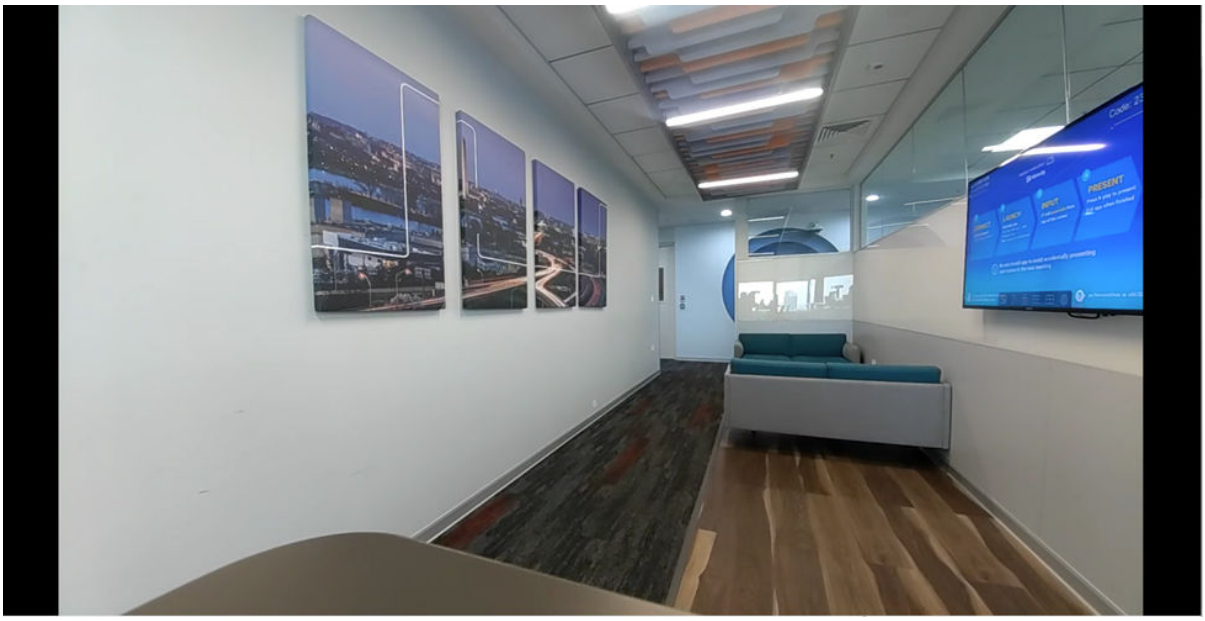


Figure 3-15 Expected output for gst-jpg-image-decode.py application–Preview

Pipeline flow

The table lists the plugins used to run the decode JPEG images pipeline:

Plugin	Description
multifilesrc	Reads the video data from sequentially named files.
capsfilter	Enforces constraints on the video data.
jpegdec	Decodes the JPEG video stream.
videoconvert	Converts the video frames from one format to another.
Waylandsink	Displays the video stream on the Wayland display.

3.10 Object detection and classification

The **gst-camera-two-stream-detection-and-classification-side-by-side.py** application uses a YOLOv8 LiteRT model to detect and classify objects in the scene displayed by the AI overlay composer.

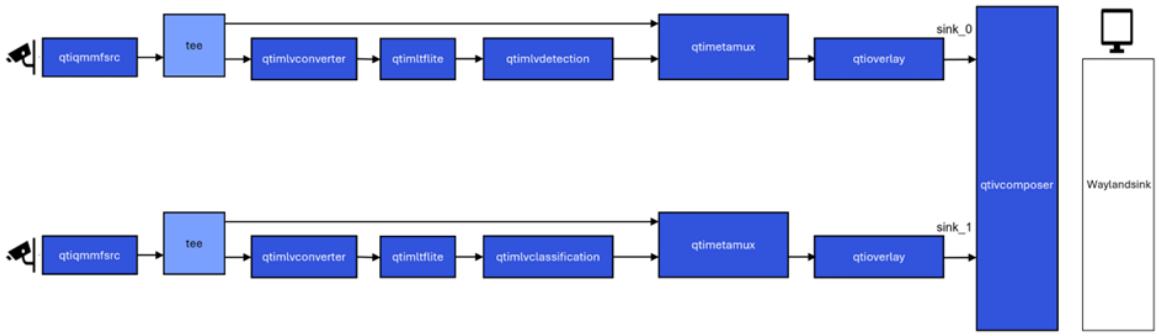


Figure 3-16 Pipeline for object detection and classification

For information about the plugins used in this pipeline, see Pipeline flow.

Model files

Table 3-5 Models used for detection and classification

Purpose	LiteRT model	Description
Object detection	YOLOv8	1. Identify the object in a scene from a camera stream. 2. Overlay the bounding boxes over the detected objects.
Image classification	Resnet101	1. Classify a scene from a camera stream. 2. Overlay the classification labels on the screen.

Run the application

1. Ensure that you complete the Prerequisites.
2. Run the detection and classification script on the target device:

```
gst-camera-two-stream-detection-and-classification-side-by-side.py
```

To display the available help options, run the following command:

```
gst-camera-two-stream-detection-and-classification-side-by-side.py -h
```

Table 3-6 Default directories for model and label files for object detection and classification python application

Model and label files	Directory
Detection model	`/etc/models/YoloV8N_Detection_Quantized.tflite`
Detection labels	`/etc/labels/yolov8n.labels`
Classification model	`/etc/models/Resnet101_Quantized.tflite`
Classification labels	`/etc/labels/resnet101.labels`

Expected output

The images are shown side by side on the display.

Pipeline flow

Table 3-7 Pipeline processing stages for object detection and classification

Process	Description
qtiqmmfsrc	Collects two video streams from the camera: ■ Stream for detection is split using tee and sent to the following: ▫ qtimetamux to retain the video stream. ▫ qtimlvconverter to convert the video stream to input tensors for the detection inference. ■ Stream for classification is split using tee and sent to the following: ▫ qtimetamux to retain the video stream. ▫ qtimlvconverter to convert the video stream to input tensors for the classification inference.
Preprocessing	
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs preprocessing: ▫ Color conversion ▫ Scaling down/up ▫ Normalization on the stream data when the model expects the floating point values as an input 3. Converts the video stream to a tensor stream on its source pad. The classification model uses this tensor stream for inferencing.
Inferencing	
qtimltflite	1. Loads the model. 2. Modifies the graph for the chosen delegate. 3. Receives the tensor stream on its sinkpad. 4. Runs the inference and produces a tensor stream with the inference results on its source pad.
Postprocessing	

Table 3-7 Pipeline processing stages for object detection and classification (cont.)

Process	Description
qtimlvdetection	1. Receives the inference tensors from the object detection model. 2. Converts the inference tensors on its sinkpad into formats such as video or text that the multimedia plugins can process later. 3. Applies the threshold to the chosen number of results. 4. Loads the corresponding modules for detection models. In this use case, qtimlvdetection does the following: a. Loads the YOLOv8 submodule. b. Produces results as structures of text. c. Sends them to the sinkpad of qtimetamux.
qtimlvclassification	1. Receives the inference results from a classification model on its sinkpad. 2. Converts the inference tensors into formats such as video or text that the multimedia plugins can process later. 3. Applies the threshold to the chosen number of results. 4. Loads the corresponding modules for the classification models. In this use case, qtimlvclassification does the following: a. Loads the submodule of the model. b. Produces results as video frames with classification labels. c. Sends them to the sinkpad of qtivcomposer.
qtimetamux	1. Receives video stream and text stream with bounding box results corresponding to the video stream on its sinkpads. 2. Produces GST buffers with contents of the video stream from its sink pad. 3. Adds bounding boxes as GstVideoRegionOfInterest from data sinkpad to GST buffers meta (meta muxing) on its source pad.
qtivoverlay	1. Receives the multiplexed stream. 2. Overlays the bounding boxes on the VideoFrame using CL. 3. Produces GST buffers with overlays in its source pad.
qtivcomposer	1. Receives the original video stream with classification results on its sinkpads. 2. On its sourcepad, produces GST buffers with contents composed of video streams from its sinkpads.
Output	
Waylandsink	1. Receives the video in its sinkpad 2. Submits the video stream to Weston. 3. Weston renders the video stream on a local display device.

RELATED INFORMATION

"Object detection" on page 33
"Image classification" on page 28

3.11 Transform and encode a camera stream

The **gst-camera-rotate-downscale-file.py** application rotates, downscales, and encodes a camera stream. The transformed stream is then saved to a file.

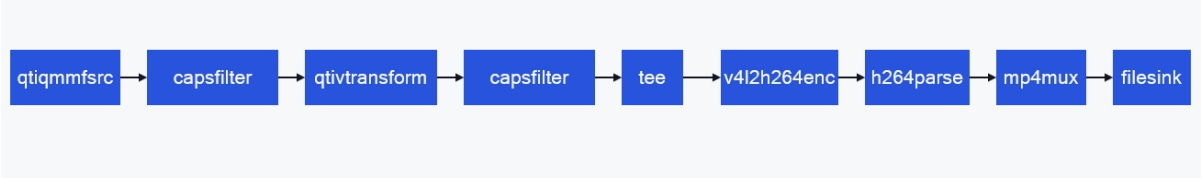


Figure 3-17 Pipeline for video transformation

For information about the plugins used in this pipeline, see Pipeline flow.

Run the application

1. Ensure that you complete the Prerequisites.
2. Run the transformation script on the target device:

   ```
   gst-camera-rotate-downscale-file.py
   ```

To display the available help options, run the following command:

```
gst-camera-rotate-downscale-file.py -h
```

Expected output

The output is saved to a file at `/etc/media/test.mp4`.

Pipeline flow

The table lists the plugins used to run the transformation pipeline:

Plugin	Description
qtiqmmfsrc	Captures the video stream.
qtivtransform	Transforms the video stream.
v4l2h264enc	Encodes H.264 video.
h264parse	Parses the encoded H.264 video stream.
mp4mux	Multiplexes the stream into an MP4 container.
filesink	Writes the video data to a file.

RELATED INFORMATION

"Video transformation" on page 166

3.12 Camera encode, object detection, and display

The **gst-camera-two-stream-encode-file-detection-display.py** application encodes the camera stream and saves it to a file. The application uses a YOLOv8 LiteRT model to identify the objects in a scene from a camera stream. The application overlays the bounding boxes over the detected objects and displays the results.

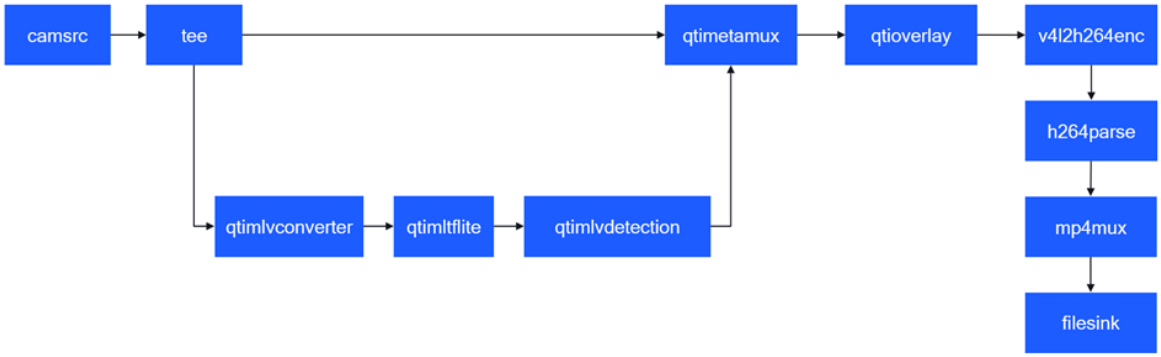


Figure 3-18 Pipeline for camera encode and object detection

For information about the plugins used in this pipeline, see Pipeline flow.

Run the application

1. Ensure that you complete the Prerequisites.
2. Run the camera encode and object detection script on the target device:

   ```
   gst-camera-two-stream-encode-file-detection-display.py
   ```

To display the available help options, run the following command:

```
gst-camera-two-stream-encode-file-detection-display.py -h
```

The following are the default files in the Python script:

Table 3-8 Default model and label files for gst-camera-two-stream-encode-file-detection-display.py

Files	Directory
Detection model (YOLOv8)	`/etc/models/YoloV8N_Detection_Quantized.tflite`
Detection labels (same for both models)	`/etc/labels/yolov8n.labels`

Expected output

The output is saved at `/etc/media/test.mp4`.

Pipeline flow

Table 3-9 Pipeline processing stages for camera encode and object detection

Process	Description
qtiqmmfsrc	Collects two video streams from the camera: ▪ One stream is saved to a file. ▪ The second stream is used for detection. It's split using tee and sent to the following: ▫ qtimetamux to retain the video stream. ▫ qtimlvconverter to convert the video stream to input tensors for the classification inference.
v4l2h264enc	Encodes H.264 video.
h264parse	Parses H.264 video.
mp4mux	Multiplexes the video data.
filesink	Saves the video data to a file.
Preprocessing	
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs preprocessing: ▫ Color conversion ▫ Scaling down/up ▫ Normalization on the stream data when the model expects the floating point values as an input 3. Converts the video stream to a tensor stream on its source pad. The object detection model uses this tensor stream for inferencing.
Inferencing	
qtimltflite	1. Loads the model. 2. Modifies the graph for the chosen delegate. 3. Receives the tensor stream on its sinkpad. 4. Runs the inference and produces a tensor stream with the inference results on its source pad.
Postprocessing	
qtimlvdetection	1. Receives the inference tensors from the object detection model. 2. Converts the inference tensors on its sinkpad into formats such as video or text that the multimedia plugins can process later. 3. Applies the threshold to the chosen number of results. 4. Loads the corresponding modules for detection models. In this use case, qtimlvdetection does the following: a. Loads the YOLOv8 submodule. b. Produces results as structures of text. c. Sends them to the sinkpad of qtimetamux.
qtimetamux	1. Receives video stream and text stream with bounding box results corresponding to the video stream on its sinkpads. 2. Produces GST buffers with contents of the video stream from its sink pad. 3. Adds bounding boxes as GstVideoRegionOfInterest from data sinkpad to GST buffers meta (meta muxing) on its source pad.

Table 3-9 Pipeline processing stages for camera encode and object detection (cont.)

Process	Description
qtivoverlay	1. Receives the multiplexed stream. 2. Overlays the bounding boxes on the VideoFrame using CL. 3. Produces GST buffers with overlays in its source pad.
Output	
Waylandsink	1. Receives the video in its sinkpad 2. Submits the video stream to Weston. 3. Weston renders the video stream on a local display device.

RELATED INFORMATION

"Object detection" on page 33

3.13 Object detection, classification, and segmentation

The **gst-filesrc-2detection-classification-segmentation-side-by-side.py** script identifies an object from a scene in a camera stream, overlays the bounding boxes over the detected objects, classifies scenes from the video stream, and produces semantic segmentation for the video. The output is displayed side by side on a screen.

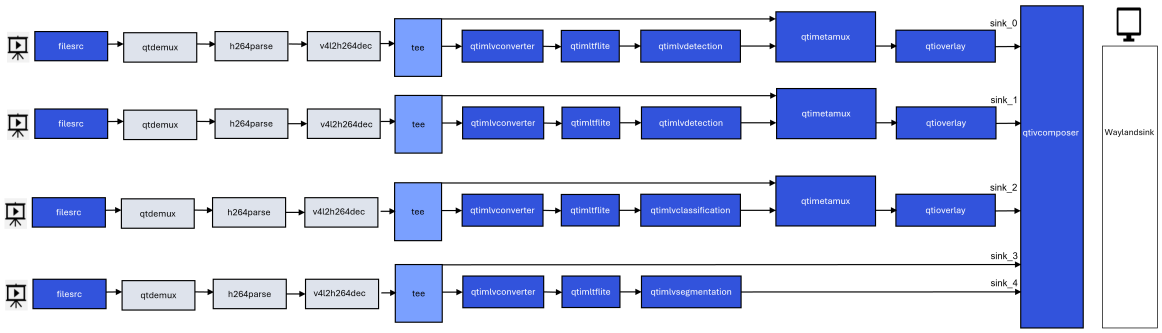


Figure 3-19 Pipeline for object detection, image classification, and segmentation

For information about the plugins used in this pipeline, see Pipeline flow.

Model files

Table 3-10 Models used for detection and classification

Purpose	LiteRT model	Description
Object detection	YOLOv8	1. Identify the object in a scene from a camera stream. 2. Overlay the bounding boxes over the detected objects.
Image classification	Resnet101	1. Classify a scene from a camera stream. 2. Overlay the classification labels on the screen.
Image segmentation	FFNet40S	Produce semantic segmentations for the video file.

Run the application

1. Ensure that you complete the Prerequisites.
2. Run the detection, classification, and segmentation script on the target device:

   ```
   gst-filesrc-2detection-classification-segmentation-side-by-side.py
   ```

To display the available help options, run the following command:

```
gst-filesrc-2detection-classification-segmentation-side-by-side.py -h
```

The following are the input videos:

Input video	Directory
Object detection	`/etc/media/detection_input.mp4`
Image classification	`/etc/media/classification_input.mp4`
Image segmentation	`/etc/media/segmentation_input.MOV`

The following are the default file paths in the Python script.

Table 3-11 Default directories for model and label files

Model and label files	Directory
Detection model	`/etc/models/yolov8_det_quantized.tflite`
Detection labels	`/etc/labels/yolov8n.labels`
Classification model	`/etc/models/Resnet101_Quantized.tflite`
Classification labels	`/etc/labels/resnet101.labels`
Segmentation model	`/etc/models/ffnet_40s_quantized.tflite`
Segmentation labels	`/etc/labels/dv3-argmax.labels`

Expected output

The four streams can be previewed side by side on a local display.

Pipeline flow

Table 3-12 Pipeline processing stages for object detection, image classification, and segmentation

Process	Description
filesrc	Reads the video data from a file.
qtdemux	Demultiplexes the video data.
h264parse	Parses the H.264 video.
v4l2h264dec	Decodes the H.264 video.
Preprocessing	

Table 3-12 Pipeline processing stages for object detection, image classification, and segmentation (cont.)

Process	Description
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs preprocessing: ▫ Color conversion ▫ Scaling down/up ▫ Normalization on the stream data when the model expects the floating point values as an input 3. Converts the video stream to a tensor stream on its source pad. The object detection, classification, and segmentation models use this tensor stream for inferencing.
Inferencing	
qtimltflite	1. Loads the model. 2. Modifies the graph for the chosen delegate. 3. Receives the tensor stream on its sinkpad. 4. Runs the inference and produces a tensor stream with the inference results on its source pad.
Postprocessing	
qtimlvdetection	1. Receives the inference tensors from the object detection model. 2. Converts the inference tensors on its sinkpad into formats such as video or text that the multimedia plugins can process later. 3. Applies the threshold to the chosen number of results. 4. Loads the corresponding modules for detection models. In this use case, qtimlvdetection does the following: a. Loads the YOLOv8 submodule. b. Produces results as structures of text. c. Sends them to the sinkpad of qtimetamux.
qtimlvclassification	1. Receives the inference results from a classification model on its sinkpad. 2. Converts the inference tensors into formats like video or text that the multimedia plugins can process later. 3. Applies the threshold to the chosen number of results. 4. Loads the corresponding modules for the classification models. In this use case, qtimlvclassification does the following: a. Loads the submodule of the model. b. Produces results as video frames with classification labels. c. Sends them to the sinkpad of qtivcomposer.
qtimlvsegmentation	1. Receives the inference tensors on its sinkpad. 2. Converts the inference tensors into video formats that the multimedia plugins can process later. 3. Produces the semantic segmentations for the frame. 4. Loads the corresponding modules for the segmentation models. In this use case, qtimlvsegmentation does the following: a. Loads the deeplab-argmax submodule. b. Produces video frames with segmentation masks. c. Sends them to the sinkpad of qtivcomposer.

Table 3-12 Pipeline processing stages for object detection, image classification, and segmentation (cont.)

Process	Description
qtimetamux	1. Receives video stream and text stream with bounding box results corresponding to the video stream on its sinkpads. 2. Produces GST buffers with contents of the video stream from its sink pad. 3. Adds bounding boxes as GstVideoRegionOfInterest from data sinkpad to GST buffers meta (meta muxing) on its source pad.
qtivoverlay	1. Receives the multiplexed stream. 2. Overlays the bounding boxes on the VideoFrame using CL. 3. Produces GST buffers with overlays in its source pad.
qtivcomposer	1. Receives the original video stream with classification results on its sinkpads. 2. On its sourcepad, produces GST buffers with contents composed of video streams from its sinkpads.
Output	
Waylandsink	1. Receives the video in its sinkpad 2. Submits the video stream to Weston. 3. Weston renders the video stream on a local display device.

RELATED INFORMATION

"Object detection" on page 33
"Image classification" on page 28
"Image segmentation" on page 42

3.14 Parallel inference using Python

The **gst-parallel-inference.py** application receives a video input from a camera, file, or an RTSP stream and sends it for a four-channel parallel processing by AI models (classification, object detection, pose detection and segmentation). The output is displayed as a preview with the overlaid AI models.

NOTE This application isn't supported on QCS8275.

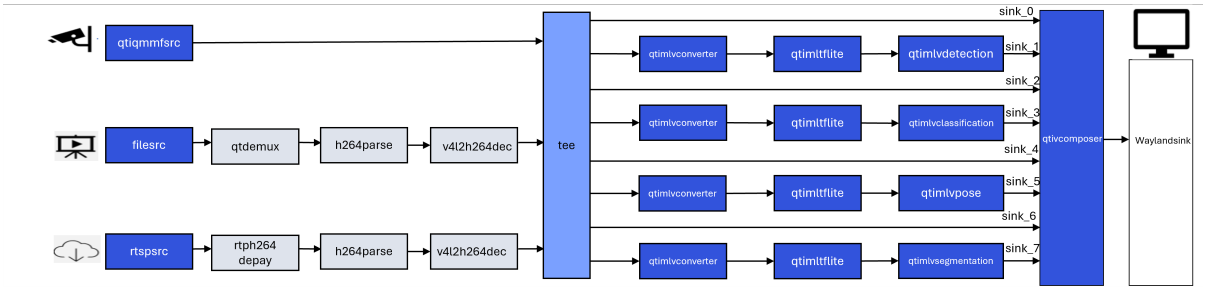


Figure 3-20 gst-parallel-inference.py pipeline

For information about the plugins used in this pipeline, see Pipeline flow.

Model and label files

The table lists the model and label files that should be available on the device before running the application.

Table 3-13 Default model and label files for gst-parallel-inference.py

Inference	Model directory	
Object detection	`/etc/models/YOLOv8-Detection-Quantized.tflite`	`/etc/label`
Pose estimation	`/etc/models/hrnet_pose_quantized.tflite`	`etc/labels` `posenet_mo`
Segmentation	`/etc/models/deeplabv3_plus_mobilenet_quantized.tflite`	`/etc/label`
Classification	`/etc/models/inception_v3_quantized.tflite`	`/etc/label`

Run the application

1. Ensure that you complete the Prerequisites.
2. Ensure that the Model and label files are available on the device.
3. Rename the label files on the EVK:

   ```
   cp /etc/labels/yolonas.labels /etc/labels/yolov8.labels
   ```

4. Run the use cases:
   - Input from the camera:

     ```
     gst-parallel-inference.py --camera
     ```

   - Input from a file:

     ```
     gst-parallel-inference.py --file "/etc/media/video.mp4"
     ```

   - Input from an RTSP stream:

     ```
     gst-parallel-inference.py --rtsp "rtsp://<ip>:<port>/<stream>"
     ```

   - Input from a camera with object detection constants:

     ```
     gst-parallel-inference.py --camera --constants-detection "YOLOv8,q-
     offsets=<21.0, 0.0, 0.0>,q-scales=<3.093529462814331, 0.00390625, 1.0>;"
     ```

   - Input from a camera with classification constants:

     ```
     gst-parallel-inference.py --camera --constants-classification
     "Mobilenet,q-offsets=<38.0>,q-scales=<0.15008972585201263>;"
     ```

   - Input from a camera with pose detection constants:

     ```
     gst-parallel-inference.py --camera --constants-pose "Posenet,q-
     offsets=<8.0>,q-scales=<0.0040499246679246426>;"
     ```

   - Input from a camera with segmentation constants:

     ```
     gst-parallel-inference.py --camera --constants-segmentation "deeplab,q-
     offsets=<0.0>,q-scales=<1.0>;"
     ```

To display the available help options, run the following command:

```
gst-parallel-inference.py -h
```

Expected output

After performing the four parallel inferences, the results are displayed side by side on the screen.

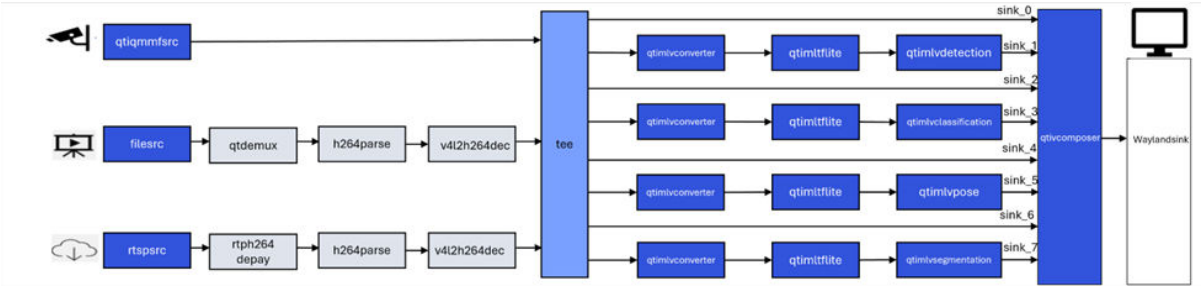


Figure 3-21 Expected output for gst-ai-parallel-inference.py application

Pipeline flow

The table lists the plugins used in the parallel inference pipeline:

Plugin	Description
Camera source:qtiqmmfsrc	▪ Captures the live stream from camera. ▪ Uses tee to split the stream for inferencing.
File source: filesrc	▪ Captures the video stream using filesrc, followed by qtdemux, which demultiplexes the stream. ▪ Uses tee to split the stream for inferencing.
RTSP source: rtspsrc	▪ Captures the RTSP stream using rtspsrc, followed by rtph264depay for video extraction. ▪ Uses tee to split the stream for inferencing.
h264parse	Parses the H.264 video.
v4l2h264dec	Decodes the video
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs the following preprocessing on the stream data. This preprocessing is done when the model expects floating-point values as input. a. Color conversion b. Scaling (up or down) c. Normalization The tensor stream is used for inferencing in the later stages of the pipeline.
qtimltflite	1. After the inference runtime receives the tensor stream on its sink pad, it runs the inference. 2. Produces a tensor stream with the inference results on its source pad.
Postprocessing plugins	Handles the inference results from any object detection, classification, pose detection, and segmentation model. ▪ qtimlvdetection a. Applies a threshold to the chosen number of results. b. Loads the YOLOv8 module. c. Produces video frames with only bounding boxes that can be overlaid on objects, sending them to the sink pad of the qtivcomposer.

Plugin	Description
	■ qtimlvclassification a. Applies the threshold to the chosen number of results. b. Loads the MobileNet module. c. Produces results as video frames with classification labels, sending them to the sink pad of the qtivcomposer. ■ qtimlvpose a. Applies the threshold to the chosen number of results. b. Loads the corresponding modules for various pose estimation models. For the use cases described in this section, qtimlvpose loads the PoseNet module. c. Produces results as video frames with poses drawn, sending them to the sink pad of the qtivcomposer. ■ qtimlvsegmentation: Converts the inference tensors that it receives on its sink pad into video formats that the multimedia plugins for further processing.
qtivcomposer	1. Composes frames with contents from its sink pads. 2. Pushes the GStreamer buffers containing these composed frames to its source pad.
Waylandsink	1. Waylandsink submits the video stream received on its sink pad to Weston. 2. Weston renders the video stream on a local display.

RELATED INFORMATION

"Parallel inferencing" on page 50

3.15 Daisy chain detection and pose detection using Python

The **gst-ai-daisychain-detection-pose.py** application allows you to perform cascaded object detection and pose detection with input from a camera, file source, or an RTSP stream.

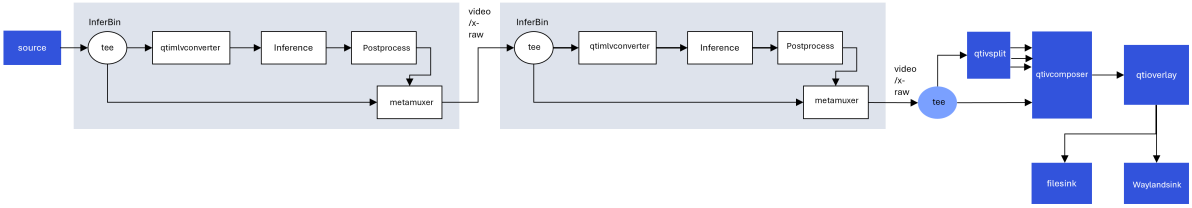


Figure 3-22 gst-ai-daisychain-detection-pose.py pipeline

For information about the plugins used in this pipeline, see Pipeline flow.

Run the application

1. Ensure that you complete the Prerequisites.
2. Rename the label files on the Linux host:

```
cp /etc/labels/yolonas.labels /etc/labels/yolov8.labels
```

3. Run the use cases:
   - Input from a camera:

```
gst-daisychain-detection-pose.py  --camera
```

   - Input from a file:

```
gst-daisychain-detection-pose.py  --file /etc/media/video.mp4
```

- Input from an RTSP stream:

```
gst-daisychain-detection-pose.py  --rtsp "rtsp://<ip>:<port>/<stream>"
```

- Input from a camera with detection constants:

```
gst-daisychain-detection-pose.py  --camera --constants-detection
"YOLOv8,q-offsets=<21.0, 0.0, 0.0>,q-scales=<3.093529462814331,
0.00390625, 1.0>;"
```

- Input from a camera with pose constants:

```
gst-daisychain-detection-pose.py  --camera --constants-pose "hrnet,q-
offsets=<8.0>,q-scales=<0.0040499246679246426>;"
```

To display the available help options, run the following command:

```
gst-daisychain-detection-pose.py -h
```

Expected output

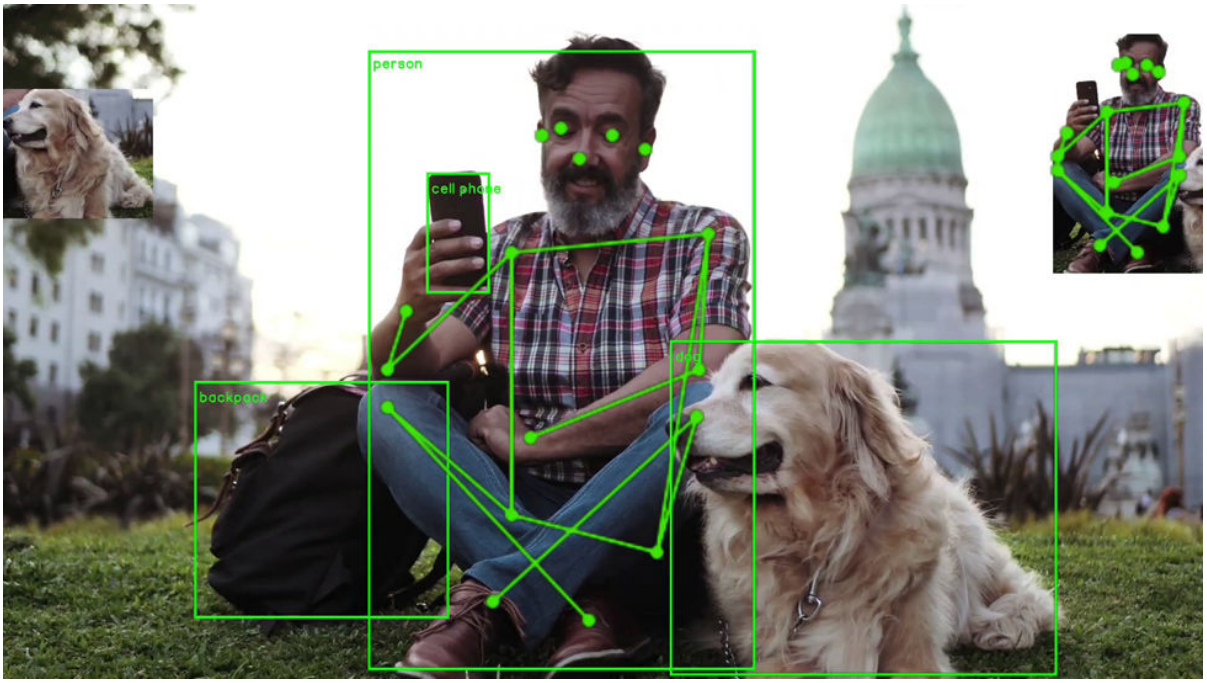


Figure 3-23 Expected output for gst-ai-daisychain-detection-pose.py application

Pipeline flow

The table lists the plugins used in the daisy chain detection and pose estimation pipeline:

Plugin	Description
Camera source:qtiqmmfsrc	▪ Captures the live stream from camera. ▪ Uses tee to split the stream for inferencing.
File source: filesrc	▪ Captures the video stream using filesrc, followed by qtdemux, which demultiplexes the stream. ▪ Uses tee to split the stream for inferencing.

Plugin	Description
RTSP source: rtspsrc	▪ Captures the RTSP stream using rtspsrc, followed by rtph264depay for video extraction. ▪ Uses tee to split the stream for inferencing.
qtimetamux	Multiplexes the stream.
h264parse	Parses the H.264 video.
v4l2h264dec	Decodes the H.264 video stream.
qtivsplit	Crops full frame into smaller frames based on the detected bounding boxes detected (maximum 4).
qtimlvconverter	Used by AI processing stream for preprocessing: 1. Receives the video stream on its sink pad. 2. Performs the following preprocessing on the stream data. This preprocessing is done when the model expects floating-point values as input. a. Color conversion b. Scaling (up or down) c. Normalization 3. Converts the preprocessed video stream to a tensor stream on its source pad. The tensor stream is used for inferencing in the later stages of the pipeline.
qtimltflite	Runs on LiteRT and uses the yolov5.tflite model for object detection and `inception_v3_quantized.tflite` for classification. 1. After the inference runtime receives the tensor stream on its sink pad, it runs the inference. 2. Produces a tensor stream with the inference results on its source pad.
qtimlvdetection	Handles inference results from any classification model. 1. Applies a threshold to the chosen number of results. 2. Loads the YOLOv5 module. 3. Generates results in the form of video frames with classification labels. 4. Produces video frames with only bounding boxes that can be cropped.
qtimlvpose	▪ Applies a threshold to the chosen number of results. ▪ Loads corresponding modules for various pose detection models. In the use case described in this section, qtimlvpose does the following: a. Loads the HRNet module. b. Produces results in the form of video frames with drawn poses. c. Sends the results to the sink pad of qtivcomposer for further processing or display.
qtivcomposer	1. Composes frames by combining the contents from its sink pads. 2. Pushes the GStreamer buffers containing the composed frames onto its source pad.
Waylandsink	1. Forwards the video stream received on its sink pad to Weston. 2. Weston renders the video stream on a local display.

Sample model and label files

Table 3-14 Sample model and label files for gst-ai-daisychain-detection-pose

Runtime	Model files	Label files
LiteRT	▪ Detection: *yolov8_det_quantized.tflite* ▪ Pose: *hrnet_pose_quantized.tflite*	▪ Detection: *yolov8.labels* ▪ Pose: *hrnet_pose.labels*

RELATED INFORMATION

"Daisy chain detection and pose estimation" on page 65

3.16 Customize a Python sample application

You can develop custom sample applications using the instructions in this procedure.

The Camera encoding sample application has been used in this procedure as an example.

1. Pull the python application source code from the reference platform to host PC.

```
scp root@<IP address of target device>:/usr/bin/gst-camera-encode.py>
<destination directory>
```

2. As required, perform your customizations.
3. Push the customized source code to the device.

```
scp <filename> root@<IP address of target device>:/<directory>/
```

4. Run the sample application.

   For example:

```
python3 /<pushed-file-directory>/gst-camera-encode.py
```

4 Deploy Qualcomm IM SDK as a container

Build Qualcomm IM SDK within the docker container and prepare the final device docker image, which can be deployed to the Qualcomm Linux enabled Edge device. This allows you to use the AI workflow microservices to build various types of AI pipelines using Qualcomm IM SDK.

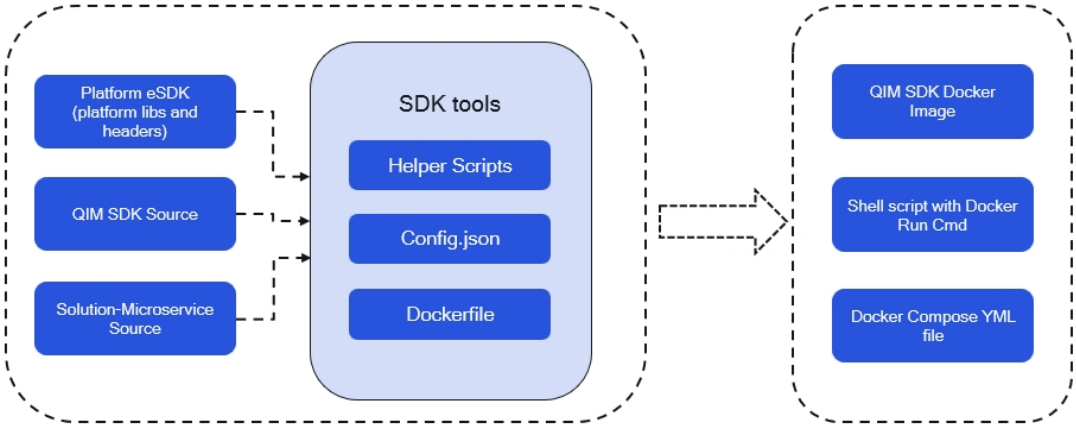


Figure 4-1 Qualcomm IM SDK within a container - architecture

The table lists the components that are required to build the Qualcomm IM SDK docker container from the source code:

Table 4-1 Components for Qualcomm IM SDK docker container

Component	Description
Platform eSDK	▪ Provides the platform dependencies for Qualcomm IM SDK. ▪ Contains platform libraries and header files.
Qualcomm IM SDK source code	Contains the source code of the plugins and recipe files, which build and define the compile time dependencies.
solutions-microservices	Provides the AI pipelines that are required for microservices.
sdk-tools	▪ Provides the docker files and a set of helper scripts to generate the Qualcomm IM SDK device docker container. ▪ Provides the *run* commands for the reference docker, and the YAML compose files to run the docker containers. ▪ Maps all the required platform devices and libraries to provide hardware acceleration inside the docker container.

4.1 Build Qualcomm IM SDK docker image

1. Create a root workspace:

   ```
   mkdir <your workspace>/iot-solutions
   ```

   This root workspace has the following:

   - Extracted platform eSDK
   - `meta-qti-gst`
   - `gst-plugins-qti-oss`
   - `solutions-microservices`
   - `sdk-tools`
   - `le-services`

2. Get the Platform eSDK.

   The Platform eSDK provides the following:

   - A collection of platform libraries for a specialized environment.
   - Cross-development toolchain and libraries that are tailored to a specific Yocto-based Qualcomm Linux release.

   You can either use a prebuilt Platform eSDK or generate your own. The table guides you to the specific instructions to do both:

Use Platform eSDK	Instructions
Prebuilt Platform eSDK	Download and install Platform eSDK
Generate your own Platform eSDK	Generate an eSDK

3. Clone the following projects:

   a. Qualcomm IM SDK projects:

   ```
   # Clone QIM SDK plugins source

   cd <your workspace>/iot-solutions
   git clone https://git.codelinaro.org/clo/le/platform/vendor/qcom-
   opensource/gst-plugins-qti-oss.git -b imsdk.lnx.2.0.0.r2-rel
   cd gst-plugins-qti-oss
   git fetch origin imsdk.lnx.2.0.0.r2-rel
   git checkout FETCH_HEAD

   # Clone QIM SDK recipes
   cd <your workspace>/iot-solutions
   git clone https://git.codelinaro.org/clo/le/meta-qti-gst.git -b
   imsdk.lnx.2.0.0.r2-rel
   cd meta-qti-gst
   git fetch origin imsdk.lnx.2.0.0.r2-rel
   git checkout FETCH_HEAD
   ```

b. Solutions-Microservices project:

```
cd <your workspace>/iot-solutions
git clone https://git.codelinaro.org/clo/le/solutions-microservices.git
-b iot-solutions.lnx.1.0
cd solutions-microservices
git fetch origin iot-solutions.lnx.1.0
git checkout FETCH_HEAD
```

c. SDK-tools project:

```
cd <your workspace>/iot-solutions
git clone https://git.codelinaro.org/clo/le/sdk-tools.git -b imsdk-
tools.lnx.1.0
cd sdk-tools
git fetch origin imsdk-tools.lnx.1.0
git checkout FETCH_HEAD
```

d. Linux Embedded (LE) services project:

```
cd <your workspace>/iot-solutions
git clone https://git.codelinaro.org/clo/le/platform/vendor/qcom-
opensource/le-services.git --single-branch -b le-services.lnx.1.0.r1-rel
cd le-services
git fetch origin le-services.lnx.1.0.r1-rel
git checkout FETCH_HEAD
```

4. Configure the build:

NOTE If you are building the Qualcomm IM SDK docker build for the first time, set up the docker on your machine.

Update the JSON configuration file for the required target. Go to the `targets` directory.

```
cd <your workspace>/iot-solutions/sdk-tools/qimsdk/targets
```

The figure shows the fields in the `config.json` file:

```
{
    "Additional_tag_container": "",
    "Additional_tag_image": "",
    "Docker_image_path": "/local/mnt/workspace/iot-solutions/Docker_image_path",
    "Target_device_ID" : "<adb-device-id>",
    "Supported_targets": [
        "qcs6490",
        "qcs9100",
        "qcs8300"
    ],
    "IM_SDK_Source_Dir" : "/local/mnt/workspace/iot-solutions/gst-plugins-qti-oss",
    "IM_SDK_Meta_Dir" : "/local/mnt/workspace/iot-solutions/meta-qti-gst",
    "Solution_Microservices_Dir": "/local/mnt/workspace/iot-solutions/solutions-microservices",
    "LE_Services_Source_Dir": "/local/mnt/workspace/iot-solutions/le-services",

    "Path_to_eSDK_dir" : "/local/mnt/workspace/Platform_eSDK",

    "MAP_sources_to_dev_container" : "FALSE"
}
```

The table provides the field description for the `config.json` file:

Field	Mandatory/ Optional	Description
`"Additional_tag_container"`	Optional	Append this string to the docker image name.
`"Additional_tag_image"`	Optional	The docker image tag.
`"Docker_image_path"`	Mandatory	The directory to which the docker image and docker run script are exported.
`"Target_device_ID"`	Optional	The device ID. If the device isn't attached to a computer, this field can be blank.
`"IM_SDK_Source_Dir"`	Mandatory	The directory containing the Qualcomm IM SDK GSt plugin source code.
`"IM_SDK_Meta_Dir"`	Mandatory	The directory containing the Qualcomm IM SDK GSt plugin BitBake recipe folder.
`"Solution_Microservices_Dir"`	Mandatory	The solutions-microservices directory, which has all the Qualcomm IM SDK micro-service shell scripts.
`"LE_Services_Source_Dir"`	Mandatory	The LE-services directory, which has all the LE source files.
`"Path_to_eSDK_dir"`	Mandatory	The directory to which the Platform eSDK is extracted.

5. Build and export the docker image:
   a. Set the environment:

      ```
      cd <your workspace>/sdk-tools/qimsdk
      source scripts/docker_env_setup.sh
      ```

   b. Build the docker image:

      ```
      qimsdk-docker-build-image targets/config.json
      ```

   c. Export the docker image and the docker run scripts to the `Docker_image_path` directory.

      ```
      qimsdk-docker-device-save-image targets/config.json
      ```

4.2 Load and run the docker

1. Install the docker image on the device:

   ```
   ssh root@<device-ip>
   ```

   The password is `oelinux123`.

2. Copy the docker image from Linux host to the target device:

   ```
   scp <Docker_image_path>/* root@<device-ip>:/opt/docker
   ```

   `<Docker_image_path>` is the directory where the docker image TAR file and scripts are installed.

3. Verify whether the docker directory configuration is successful:

   ```
   docker info | grep Storage
   ```

4. Load the docker image and run the docker in the device shell:

```
cd /opt/docker
docker load -i /opt/docker/<qimsdk-docker-image>
mv docker-compose-qcs6490.yml docker-compose.yml
chmod 777 docker_run_qcs6490.sh
```

   `<qimsdk-docker-image>` is the name of the file that has the Qualcomm IM SDK docker image. For example, `qimsdk.tar`.

5. Create the docker directory in the device shell:

```
mkdir -p /opt/docker
chmod -R 777 /opt/docker
```

4.3 Run docker image for multimedia/AI use cases

NOTE The following procedure is applicable to QCS6490, QCS9075, and QCS8275. As an example, QCS6490 is used in the commands.

Prerequisites

1. Build Qualcomm IM SDK docker image.
2. Load and run the docker.

.

Procedure

A docker run command is available in the shell script.

Run the Qualcomm IM SDK docker image using the following commands:

- a. In a device shell, run the docker script:

```
sh /opt/docker/docker_run_qcs6490.sh
```

  b. List all the running containers:

```
docker ps -a
```

  c. Enter the Qualcomm IM SDK container

```
docker exec -it <qimsdk-docker-image-name> bash
```

Use docker compose commands (optional)

You can use the docker compose commands to run several docker containers in different consoles.

Do the following to run the docker compose commands:

- a. In a device shell, go to the docker directory:

```
cd /opt/docker
```

b. Run the docker-compose bash script:

```
docker-compose run <qimsdk-docker-image-name> bash
```

Field	Description
`<qimsdk-docker-image-name>`	The name of the Qualcomm IM SDK docker image.
`qimsdk`	The default docker image.

To check the action name, run the `docker images` command.

4.4 Run container use cases for camera

1. Enable the display:

```
export XDG_RUNTIME_DIR=/dev/socket/weston && export
WAYLAND_DISPLAY=wayland-1
```

2. As an example, run the use case for single 1080p YUV stream from live source:

```
gst-launch-1.0 -e qtiqmmfsrc name=camsrc video_1::type=preview ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! waylandsink fullscreen=true sync=false
```

For more use cases, see Run multimedia use cases.

Known issues

- Support for the following isn't enabled in the docker:
  - GStd.
  - NV12 UBWC compressed streams.
  - The sample applications dependent on NV12 UBWC.
- The AI USB camera application isn't supported inside the docker.
- RTSP sink isn't supported on the Multi input/output object detection application.
- Minor frame drops have been observed in the Parallel inferencing application inside the docker.
- Encoding isn't supported on the Daisy chain detection and pose estimation application inside the docker.
- The Face detection and Face recognition sample applications aren't supported inside the docker.

5 GStreamer command-line use cases

GStreamer provides command-line tools such as gst-launch-1.0 tool to help you run the AI/ML and multimedia use cases.

Prerequisites

- If not already done so, Download and install eSDK.
- To access your host device, enable SSH. For instructions, see Sign in using SSH.

  NOTE If SSH is enabled, you can skip this step.
- Push the files from the Linux host computer.

  ```
  scp <filename> root@<IP address of target device>:/etc/
  ```
- Connect the display to the device using the HDMI port. For instructions, see Set up HDMI display.
- Enable the display:

  ```
  export XDG_RUNTIME_DIR=/dev/socket/weston && export
  WAYLAND_DISPLAY=wayland-1
  ```
- Connect to the speaker to enable the audio. For instructions, see Set up audio hardware.

If you face issues while enabling camera or display, see Camera troubleshooting and Display troubleshooting.

5.1 Run machine learning use cases

LiteRT and the Qualcomm Neural Processing SDK runtime are used for inference in the machine learning use cases.

Before you run the use cases, do the following:

- Complete the preconditions mentioned in GStreamer command-line use cases.
- Download model and label files.
- Obtain model constants for `q_offset` and `q_scale` constants of the quantized LiteRT models.

IMPORTANT The AI procedures in this guide are compatible with Qualcomm AI Runtime SDK v2.32 and LiteRT (or TFLite) v2.16.1. Ensure that you download the matching SDKs to your host computer before starting AI/ML development.

The use cases described use a MobileNet LiteRT model to classify scenes from a single camera stream and either overlay or compose the classification labels.

NOTE These instructions are validated on a Ubuntu 22.04 host computer.

1. Create the Python 3.10 virtual environment:

```
sudo apt-get install python3.10
python3.10 -m venv py3.10
source py3.10/bin/activate
```

2. Generate the yolov5.tflite model:

```
git clone https://github.com/ultralytics/yolov5.git
cd yolov5
python -m pip install -r requirements.txt tensorflow-cpu
python export.py --weights yolov5m.pt --img 320 --include tflite --int8 --
data data/coco128.yaml
scp yolov5m-int8.tflite root@<IP address of the device>:/opt/yolov5.tflite
```

The YOLO-NAS and YOLOv5 models are trained on the same dataset. Use the same label file for YOLOv5.

```
ssh root@<ip-addr of the target device>a
cp /opt/yolonas.labels /opt/yolov5.labels
```

5.1.1 LiteRT use cases

LiteRT is a set of tools that allows on-device machine learning. You can run your models on mobile, embedded, and edge devices. LiteRT use cases allow you to run use cases for image classification, object detection, image segmentation, and pose estimation.

Before you run the use cases, complete the preconditions mentioned in GStreamer command-line use cases.

RELATED INFORMATION

"Configure Qualcomm GStreamer plugins" on page 343
Obtain model constants

5.1.1.1 Image classification and display with LiteRT

The use cases use the Inceptionv3 LiteRT model to classify scenes from a single camera stream and either overlay or compose the classification labels.

Use qtivoverlay plugin to apply classification overlay

Run this use case:

```
gst-launch-1.0 -e --gst-debug=2 qtiqmmfsrc name=camsrc ! video/x-
raw,format=NV12_Q08C,width=1280,height=720,framerate=30/1 ! queue ! \
tee name=split split. ! queue ! qtimetamux name=metamux ! queue !
qtivoverlay ! queue ! waylandsink fullscreen=true \
split. ! queue ! qtimlvconverter ! queue ! qtimltflite delegate=external
external-delegate-path=libQnnTFLiteDelegate.so \
external-delegate-options="QNNExternalDelegate,backend_type=htp;" model=/etc/
models/inception_v3_quantized.tflite ! queue ! \
```

```
qtimlvclassification threshold=40.0 results=2 module=mobilenet labels=/etc/
labels/classification.labels \
constants="Inceptionv3,q-offsets=<38.0>,q-scales=<0.17039915919303894>;" !
text/x-raw ! queue ! metamux.
```

To stop the use case, use **CTRL + C**.

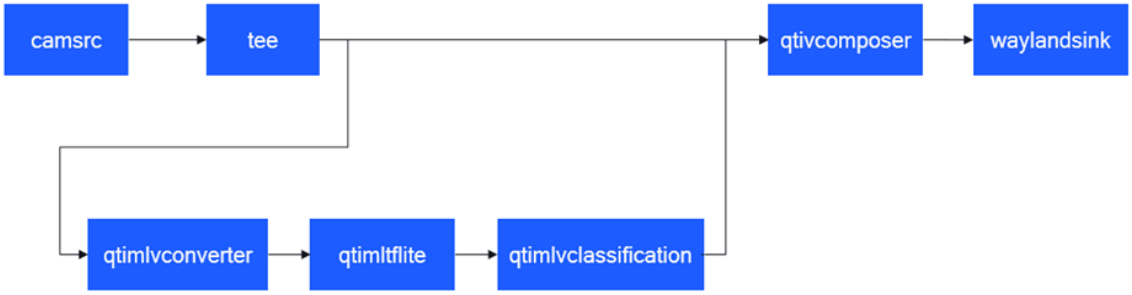


Figure 5-1 Pipeline for classification overlay

The figure shows the flow of the use case execution:

1. Classify scenes from a video stream coming through a camera source.
2. Overlay the classification labels using overlaylib.
3. Display the results.

The table provides the sequential processing stages of the pipeline execution:

Table 5-1 Pipeline processing stages for classification overlay

Process	Description
qtiqmmfsrc	1. Collects the video stream (source) and creates two copies of the source: ▫ One stream is sent to the qtimetamux plugin to retain the video stream. ▫ The other stream is sent to an ML inferencing pipeline.
Preprocessing	
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs preprocessing: ▫ Color conversion ▫ Scaling down/up ▫ Normalization on the stream data when the model expects the floating point values as an input 3. Converts the video stream to a tensor stream on its source pad. The classification model uses this tensor stream for inferencing.
Inferencing	
qtimltflite	1. Loads the model. 2. Modifies the graph for the chosen delegate. 3. Receives the tensor stream on its sinkpad. 4. Runs the inference and produces a tensor stream with the inference results on its source pad.

Table 5-1 Pipeline processing stages for classification overlay (cont.)

Process	Description
Postprocessing	
qtimlvclassification	1. Receives the inference tensors from a classification model on its sinkpad. 2. Converts the tensors into formats such as video or text that the multimedia plugins can process later. 3. Applies the threshold to the chosen number of results. 4. Loads the corresponding modules of the classification models. In this use case, qtimlvclassification does the following: a. Loads the submodule of the model. b. Produces results as structures of text. c. Sends them to the sinkpad of qtimetamux.
qtimetamux	1. Receives the video stream and text stream with classification results corresponding to the video stream on its sinkpads. 2. Produces GST buffers with the contents of video stream on its sink pad. 3. Adds classification result from data sinkpad to GST buffer meta (meta muxing) on its source pad.
qtivoverlay	1. Receives the multiplexed stream. 2. Overlays the classification labels on the VideoFrame using CL. 3. Produces GST buffers with overlays in its source pad.
Output	
Waylandsink	1. Receives the video stream on its sinkpad. 2. Submits the video stream to Weston. 3. Weston renders the video stream and possible classifications generated for that scene on a local display device.

Use qtivcomposer to mix original frame with classification mask

Run this use case:

```
gst-launch-1.0 -e --gst-debug=2 qtiqmmfsrc name=camsrc ! video/x-
raw,format=NV12_Q08C,width=1280,height=720,framerate=30/1 ! queue ! \
tee name=split split. ! queue ! qtivcomposer name=mixer
sink_1::position="<30, 30>" sink_1::dimensions="<480, 480>" ! queue !
waylandsink fullscreen=true \
split. ! queue ! qtimlvconverter ! queue ! qtimltflite delegate=external
external-delegate-path=libQnnTFLiteDelegate.so \
external-delegate-options="QNNExternalDelegate,backend_type=htp;" model=/etc/
models/inception_v3_quantized.tflite ! queue ! \
qtimlvclassification threshold=40.0 results=2 module=mobilenet labels=/etc/
labels/classification.labels \
constants="Inceptionv3,q-offsets=<38.0>,q-scales=<0.17039915919303894>;" !
video/x-raw,format=BGRA,width=640,height=480 ! queue ! mixer.
```

To stop the use case, select **CTRL + C**.

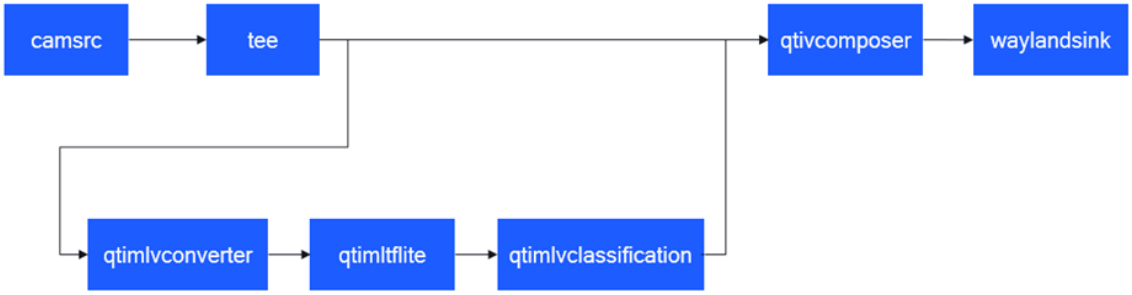


Figure 5-2 Pipeline for classification with qtivcomposer

The figure shows the flow of the use case execution:

1. Classify scenes from a video stream coming through a camera source.
2. Compose classification labels and video stream using qtivcomposer.
3. Display the results.

The table provides the sequential processing stages of the pipeline execution:

Table 5-2 Pipeline processing stages for classification with qtivcomposer

Process	Description
qtiqmmfsrc	1. Collects the video stream (source) and creates two copies of the source: ▫ One stream is sent to the qtivcomposer plugin to retain the video stream. ▫ The other stream is sent to the ML inferencing branch in the pipeline.
Preprocessing	
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs preprocessing: ▫ Color conversion ▫ Scaling down/up ▫ Normalization on the stream data when the model expects the floating point values as an input 3. Converts the video stream to a tensor stream on its source pad. The classification model uses this tensor stream for inferencing.
Inferencing	
qtimltflite	1. Loads the model. 2. Modifies the graph for the chosen delegate. 3. Receives the tensor stream on its sinkpad. 4. Runs the inference and produces a tensor stream with the inference results on its source pad.
Postprocessing	
qtimlvclassification	1. Receives the inference results from a classification model on its sinkpad. 2. Converts the inference tensors into formats like video or text that the multimedia plugins can process later.

Table 5-2 Pipeline processing stages for classification with qtivcomposer (cont.)

Process	Description
	3. Applies the threshold to the chosen number of results. 4. Loads the corresponding modules for the classification models. In this use case, qtimlvclassification does the following: a. Loads the submodule of the model. b. Produces results as video frames with classification labels. c. Sends them to the sinkpad of qtivcomposer.
qtivcomposer	1. Receives the original video stream with classification results on its sinkpads. 2. On its sourcepad, produces GST buffers with contents composed of video streams from its sinkpads.
Output	
Waylandsink	1. Receives the video in its sinkpad 2. Submits the video stream to Weston. 3. Weston renders the video stream and possible classifications generated for that scene on a local display device.

5.1.1.2 Image classification and encode with LiteRT

The use cases use the InceptionV3 LiteRT model to classify scenes from a single camera stream and either overlay or compose the classification labels, and then encode the stream.

Use qtivoverlay plugin to apply classification overlay

Run the use case:

```
gst-launch-1.0 -e qtiqmmfsrc name=camsrc ! video/x-
raw,format=NV12_Q08C,width=1280,height=720,framerate=30/1 ! queue ! tee
name=split split. ! \
queue ! qtimetamux name=metamux ! queue ! qtivoverlay ! queue ! v4l2h264enc
capture-io-mode=4 output-io-mode=5 ! \
h264parse ! queue ! mp4mux ! queue ! filesink location=/etc/media/video.mp4
split. ! queue ! qtimlvconverter ! queue ! \
qtimltflite delegate=external external-delegate-path=libQnnTFLiteDelegate.so
external-delegate-options="QNNExternalDelegate,backend_type=htp;" \
model=/etc/models/inception_v3_quantized.tflite ! queue !
qtimlvclassification threshold=40.0 results=2 module=mobilenet \
labels=/etc/labels/classification.labels constants="Inceptionv3,q-
offsets=<38.0>,q-scales=<0.17039915919303894>;" ! text/x-raw ! queue !
metamux.
```

To stop the use case, use **CTRL + C**.

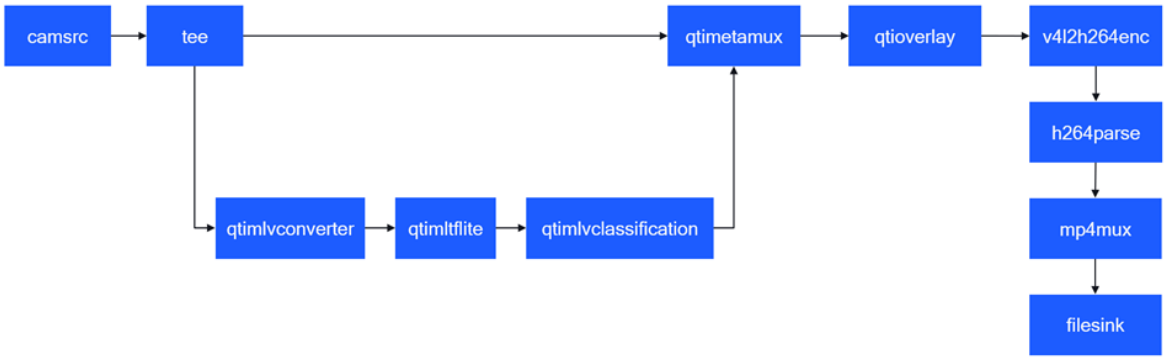


Figure 5-3 Pipeline for classification overlay and encode

The figure shows the flow of the use case execution:

1. Identify the stream coming through a camera source.
2. Overlay the classification labels using overlaylib.
3. Encode the stream as a H.264 bitstream.
4. Multiplex the stream in an MP4 container and stored as an MP4 file.

The table provides the sequential processing stages of the pipeline execution:

Table 5-3 Pipeline processing stages for classification overlay and encode

Process	Description
qtiqmmfsrc	1. The video stream is collected from a camera source plugin and two copies are created: ▫ One stream is sent to the qtimetamux plugin to retain the video stream. ▫ The other stream is sent to a ML inferencing pipeline.
Preprocessing	
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs preprocessing: ▫ Color conversion ▫ Scaling down/up ▫ Normalization on the stream data when the model expects the floating point values as input 3. Converts the video stream to a tensor stream on its source pad. The classification model uses this tensor stream for inferencing.
Inferencing	
qtimltflite	1. Loads the classification model. 2. Modifies the graph for the chosen delegate. 3. Receives the tensor stream on its sinkpad. 4. Runs the inference and produces a tensor stream with the inference results on its source pad.
Postprocessing	

Table 5-3 Pipeline processing stages for classification overlay and encode (cont.)

Process	Description
qtimlvclassification	1. Receives the inference tensors from a classification model on its sinkpad. 2. Converts the tensors into formats such as video or text that the multimedia plugins can process later. 3. Applies the threshold to the chosen number of results. 4. Loads the corresponding modules of the classification models. In this use case, qtimlvclassification does the following: a. Loads the InceptionV3 submodule. b. Produces results as structures of text. c. Sends them to the sinkpad of qtimetamux.
qtimetamux	1. Receives the video stream and text stream with classification results corresponding to video stream on its sinkpads. 2. Produces GST buffers with the contents of video stream on its sink pad. 3. Adds classification result from data sinkpad to GST buffer meta (meta muxing) on its source pad.
qtivoverlay	1. Receives the multiplexed stream. 2. Overlays the classification labels on the VideoFrame using CL. 3. Produces GST buffers with overlays in its source pad.
v4l2h264enc	1. Applies parameters to each frame of the video stream it's receiving on its sinkpad. 2. Encodes it into bitstream and sends it over its sourcepad.
h264parse	Adds more information about the bitstream to the GStreamer buffer meta.
mp4mux	Receives these buffers and creates containers with format specification buffers.
Output	
Filesink	Stores the resulting stream in a `/opt/video.mp4` file.
Playback	Pull `video.mp4` from the host computer and play it on a media player: `scp root@<IP address of target device>:/opt/ <destination directory>`

Use qtivcomposer to mix original frame with classification mask

Run the use case:

```
gst-launch-1.0 -e --gst-debug=2 qtiqmmfsrc name=camsrc ! video/x-
raw,format=NV12_Q08C,width=1280,height=720,framerate=30/1 ! queue ! tee
name=split split. ! \
queue ! qtivcomposer name=mixer sink_1::position="<30, 30>"
sink_1::dimensions="<320, 180>" ! queue ! video/x-
raw,format=NV12,width=1920,height=1080,interlace-
mode=progressive,colorimetry=bt601 ! \
v4l2h264enc capture-io-mode=4 output-io-mode=5 ! h264parse ! queue ! mp4mux !
queue ! filesink location=/etc/media/video.mp4 split. ! queue !
qtimlvconverter ! queue ! \
qtimltflite delegate=external external-delegate-path=libQnnTFLiteDelegate.so
external-delegate-options="QNNExternalDelegate,backend_type=htp;" \
model=/etc/models/inception_v3_quantized.tflite ! queue !
```

```
qtimlvclassification threshold=40.0 results=2 module=mobilenet \
labels=/etc/labels/classification.labels constants="Inceptionv3,q-
offsets=<38.0>,q-scales=<0.17039915919303894>;" ! video/x-
raw,format=BGRA,width=640,height=360 ! queue ! mixer.
```

To stop the use case, use **CTRL + C**.

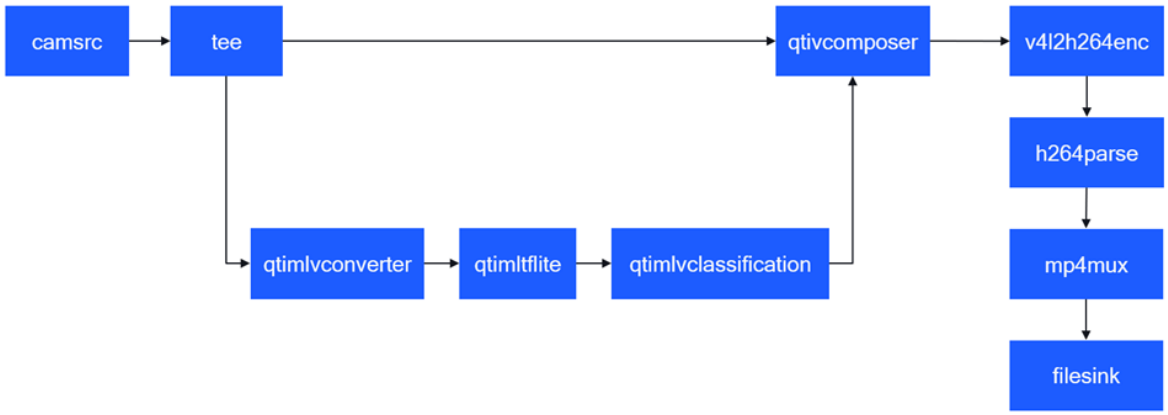


Figure 5-4 Pipeline for classification and encode with qtivcomposer

The figure shows the flow of the use case execution:

1. Identify the stream coming through a camera source.
2. Compose the classification labels and video stream using qtivcomposer.
3. Encode the stream as an H.264 bitstream.
4. Multiplex the stream in an MP4 container and store it as an MP4 file.

The table provides the sequential processing stages of the pipeline execution:

Table 5-4 Pipeline processing stages for classification and encode with qtivcomposer

Process	Description
qtiqmmfsrc	1. Collects the video stream (source) from a camera and creates two copies of the source: ▫ One stream is sent to the qtimetamux plugin to retain the video stream. ▫ The other stream is sent to an ML inferencing pipeline.
Preprocessing	
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs preprocessing: ▫ Color conversion ▫ Scaling down/up ▫ Normalization on the stream data when the model expects the floating point values as an input 3. Converts the video stream to a tensor stream on its source pad. The classification model uses the tensor stream for inferencing.
Inferencing	

Table 5-4 Pipeline processing stages for classification and encode with qtivcomposer (cont.)

Process	Description
qtimltflite	1. Loads the classification model. 2. Modifies the graph for the chosen delegate. 3. Receives the tensor stream on its sinkpad. 4. Runs the inference and produces the tensor stream with the inference results on its source pad.
Postprocessing	
qtimlvclassification	1. Receives the inference tensors from a classification model on its sinkpad. 2. Converts the tensors into formats such as video or text that the multimedia plugins can process later. 3. Applies the threshold to the chosen number of results. 4. Loads the corresponding modules of the classification models. In this use case, qtimlvclassification does the following: a. Loads InceptionV3 submodule. b. Produces results as video frames with classification labels. c. Sends them to the sinkpad of qtivcomposer.
qtivcomposer	1. Receives the original video stream with classification results on its sinkpads. 2. On its sourcepad, produces GST buffers with contents composed of video streams from its sinkpads.
v4l2h264enc	1. Applies parameters to each frame of the video stream it's receiving on its sinkpad. 2. Encodes it into bitstream and sends it over its sourcepad.
h264parse	Adds more information about the bitstream to the GStreamer buffer meta.
mp4mux	Receives these buffers and creates containers with format specification buffers.
Output	
Filesink	Stores the resulting stream in a `/etc/media/video.mp4` file.
Playback	Pull `video.mp4` from the host computer and play it on a media player: `scp root@<IP address of target device>:/etc/ <destination directory>`

5.1.1.3 Audio classification decode and display with LiteRT

The use cases use the YAMNet LiteRT model to classify and decode audio samples from a microphone and a file source.

To stop the use cases, use **CTRL + C**.

Audio classification on audio samples from microphone

Run this use case:

```
gst-launch-1.0 -v pulsesrc ! audio/x-raw,format=S16LE ! audiobuffersplit
output-buffer-size=31200 ! \
qtimlaconverter sample-rate=16000 feature=lmfe
params="params,nfft=512,nhop=160,nmels=64,chunklen=0.96;" ! queue ! \
qtimltflite model=/etc/models/yamnet.tflite  !  qtimlaclassification
```

```
module=yamnet labels=/etc/labels/yamnet.labels ! \
video/x-raw,width=640,height=360 ! queue ! waylandsink sync=false
fullscreen=true
```

The figure shows the flow of the use case execution:

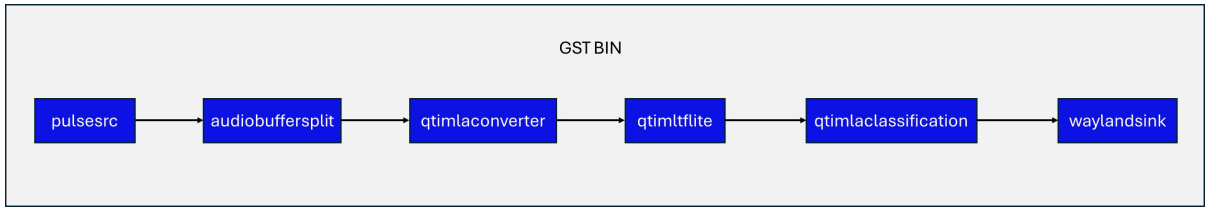


Figure 5-5 Pipeline flow for audio classification and display

The table provides the sequential processing stages of the pipeline execution:

Table 5-5 Pipeline processing stages for audio classification

Process	Description
pulsesrc	Collects the audio stream (source) from the microphone.
audiobuffersplit	Splits the incoming audio buffers into equal sized chunks.
qtimlaconverter	Performs preprocessing on the audio stream and converts the stream to a tensor stream. The audio classification model uses this tensor stream for inferencing.
qtimltflite	1. Loads the model. 2. Modifies the graph for the chosen delegate. 3. Receives the tensor stream on its sinkpad. 4. Runs the inference and produces a tensor stream with the inference results on its source pad.
qtimlaclassification	Handles the audio classification inference results: 1. Applies a threshold to the chosen number of results. 2. Creates text overlay for classes.
Waylandsink	1. Receives the video and audio streams on its sinkpad. 2. Submits the streams to Weston. 3. Weston renders the video stream and the classified audio generated for that scene on a local display device.

Audio classification and decode on audio samples from file source

- Run the use case using a FLAC decoder:

```
gst-launch-1.0 -e --gst-debug=2 filesrc location=/etc/media/
video_FLAC.mp4  ! qtdemux name=demux demux. ! queue ! h264parse ! \
v4l2h264dec  capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! qtivcomposer name=mixer  sink_1::position="<50, 50>"
sink_1::dimensions="<368, 64>" !  \
queue  ! waylandsink fullscreen=true demux. ! queue !  flacparse !
flacdec  ! queue ! audioconvert ! audio/x-raw,format=S16LE !
audioresample ! \
audiobuffersplit output-buffer-size=31200 ! queue ! qtimlaconverter
```

```
sample-rate=16000 feature=lmfe
params="params,nfft=512,nhop=160,nmels=64,chunklen=0.96;" ! \
queue ! qtimltflite name=infeng model=/etc/models/yamnet.tflite   !
qtimlaclassification name=postproc threshold=10.0  results=3 module=yamnet
\
labels=/etc/labels/yamnet.labels ! video/x-
raw,format=BGRA,width=368,height=64 ! queue ! mixer.
```

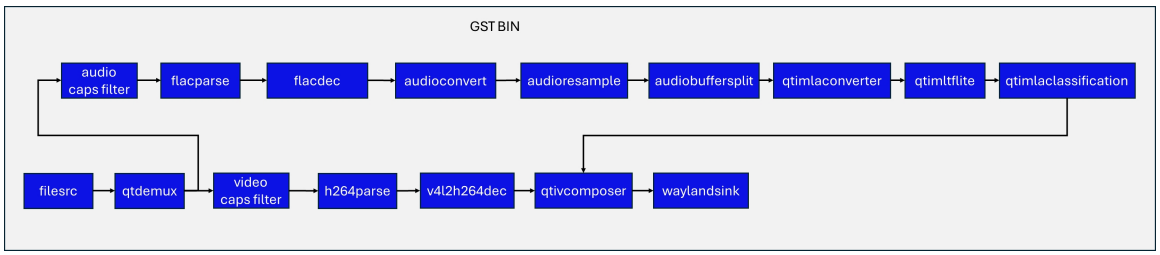


Figure 5-6 Pipeline flow for audio classification and decode–FLAC decoder

- Run the use case using the mpg123audiodec decoder:

```
gst-launch-1.0 -e --gst-debug=2 filesrc location=/etc/media/
video_mp3.mp4  ! qtdemux name=demux demux. ! queue ! h264parse !
v4l2h264dec  capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! qtivcomposer name=mixer  sink_1::position="<50, 50>"
sink_1::dimensions="<368, 64>" !  queue  ! waylandsink fullscreen=true
demux. ! queue !  mpegaudioparse ! mpg123audiodec  ! queue !
audioconvert ! audio/x-raw,format=S16LE ! audioresample !
audiobuffersplit output-buffer-size=31200 ! queue ! qtimlaconverter
sample-rate=16000 feature=lmfe
params="params,nfft=512,nhop=160,nmels=64,chunklen=0.96;" !  queue !
qtimltflite name=infeng model=/etc/models/yamnet.tflite  !
qtimlaclassification name=postproc threshold=10.0  results=3 module=yamnet
labels=/etc/labels/yamnet.labels ! video/x-
raw,format=BGRA,width=368,height=64 ! queue ! mixer.
```

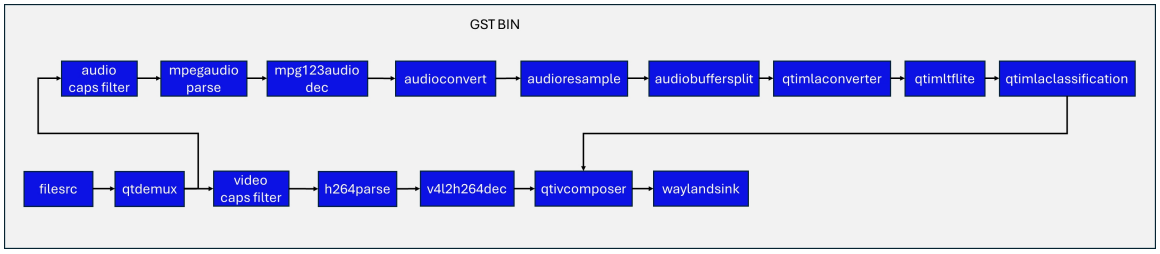


Figure 5-7 Pipeline flow for audio classification and decode–mpg123audiodec decoder

The table provides the sequential processing stages of the pipeline execution:

Table 5-6 Pipeline processing stages for audio classification using FLAC and mpg123audiodec decoders

Plugin	Description
File source: filesrc	Captures the video and audio stream, followed by qtdemux, which demultiplexes the stream.
Audio and video capsfilter	Ensures that the video and audio streams are in the correct format.
▪ Audio: mpegaudioparse or flacparse ▪ Video: h264parse	Parses the audio and the H.264 video.
▪ Audio: mpg123audiodec or flacdec ▪ Video: v4l2h264dec	Decodes the audio and video.
audioconvert	Converts the audio buffers between various possible formats.
audioresample	Resamples the audio buffers to different sample rates.
audiobuffersplit	Splits the incoming audio buffers into equal sized chunks.
qtimlaconverter	Performs preprocessing on the audio stream and converts the stream to a tensor stream. The audio classification model uses this tensor stream for inferencing.
qtimltflite	Performs inferencing using the YAMNet model.
qtimlaclassification	Handles the audio classification inference results: 1. Applies a threshold to the chosen number of results. 2. Creates text overlay for classes.
qtivcomposer	Combines the text overlay for classification results and video preview.
Waylandsink	1. Waylandsink submits the video stream received on its sink pad to Weston. 2. Weston renders the video stream on a local display.

RELATED INFORMATION

"Audio classification" on page 96

5.1.1.4 Object detection and display with LiteRT

The use cases use a YOLOv5 LiteRT model to identify the object in a scene. The use case is to either overlay or compose the bounding boxes over the detected objects, and then display the results.

Use qtivoverlay plugin to apply bounding box overlay

Run the use case:

```
gst-launch-1.0 -e --gst-debug=2 qtiqmmfsrc name=camsrc ! video/x-
raw,format=NV12_Q08C,width=1280,height=720,framerate=30/1 ! \
queue ! tee name=split split. ! queue ! qtimetamux name=metamux ! queue !
qtivoverlay ! queue ! waylandsink fullscreen=true split. ! queue ! \
```

```
qtimlvconverter ! queue ! qtimltflite delegate=external external-delegate-
path=libQnnTFLiteDelegate.so \
external-delegate-options="QNNExternalDelegate,backend_type=htp;" model=/etc/
models/yolov5.tflite ! queue ! \
qtimlvdetection threshold=75.0 results=10 module=yolov5 labels=/etc/labels/
yolov5.labels \
constants="YoloV5,q-offsets=<3.0>,q-scales=<0.005047998391091824>;" ! text/x-
raw ! queue ! metamux.
```

To stop the use case, use **CTRL + C**.

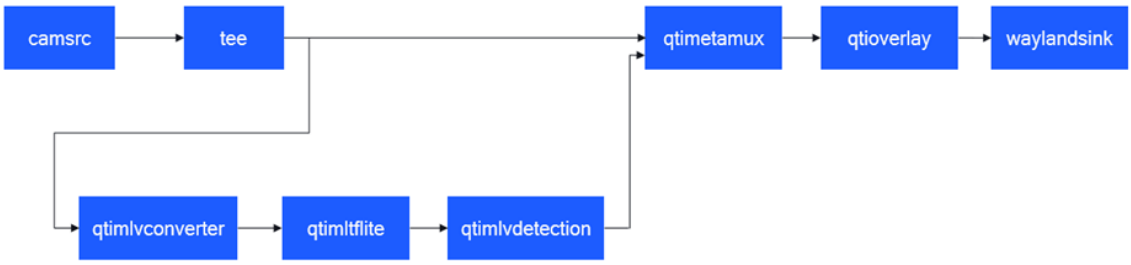


Figure 5-8 Pipeline for bounding box overlay

The figure shows the flow of the use case execution:

1. Identifies object scenes in the scene from a video stream, which is coming through a camera source.
2. Overlays bounding boxes over the detected objects using overlaylib.
3. Displays the results.

The table provides the sequential processing stages of the pipeline execution:

Table 5-7 Pipeline processing stages for bounding box overlay

Process	Description
qtiqmmfsrc	1. Collects the video stream (source) and creates two copies of the source: ▫ One stream is sent to qtimetamuxplugin to retain the video stream. ▫ The other stream is sent to an ML inferencing pipeline.
Preprocessing	
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs preprocessing: ▫ Color conversion ▫ Scaling down/up ▫ Normalization on the stream data when the model expects the floating point values as input 3. Converts the video stream to a tensor stream on its source pad. The object detection model uses this tensor stream for inferencing.
Inferencing	
qtimltflite	1. Loads the object detection model. 2. Modifies the graph for the chosen delegate.

Table 5-7 Pipeline processing stages for bounding box overlay (cont.)

Process	Description
	3. Receives the tensor stream on its sinkpad. 4. Runs the inference and produces a tensor stream with the object detection results on its source pad.
Postprocessing	
qtimlvdetection	1. Receives the inference tensors from object detection. 2. Converts the inference tensors on its sinkpad into formats like video or text that the multimedia plugins can process later. 3. Applies the threshold to the chosen number of results. 4. Loads the corresponding modules for detection models. In this use case, qtimlvdetection does the following: a. Loads the YOLOv5 submodule. b. Produces results as structures of text. c. Sends them to the sinkpad of qtimetamux.
qtimetamux	1. Receives video stream and text stream with bounding box results corresponding to the video stream on its sinkpads. 2. Produces GST buffers with contents of video stream from its sink pad. 3. Adds bounding boxes as GstVideoRegionOfInterest from data sinkpad to GST buffers meta (meta muxing) on its source pad.
qtivoverlay	1. Receives the multiplexed stream. 2. Overlays the bounding boxes on the VideoFrame using CL. 3. Produces GST buffers with overlays in its source pad.
Output	
Waylandsink	1. Receives the video stream on its sinkpad. 2. Submits the video stream to Weston. 3. Weston renders the video stream and bounding boxes generated for the objects in that scene on a local display device.

Use qtivcomposer to mix original frame with bounding box mask

Run the use case:

```
gst-launch-1.0 -e --gst-debug=2 qtiqmmfsrc name=camsrc ! video/x-
raw,format=NV12_Q08C,width=1280,height=720,framerate=30/1 ! queue ! tee
name=split split. ! \
queue ! qtivcomposer name=mixer sink_1::dimensions="<1920,1080>" ! queue !
waylandsink fullscreen=true split. ! queue ! qtimlvconverter ! queue ! \
qtimltflite delegate=external external-delegate-path=libQnnTFLiteDelegate.so
external-delegate-options="QNNExternalDelegate,backend_type=htp;" \
model=/etc/models/yolov5.tflite ! queue ! qtimlvdetection threshold=75.0
results=10 module=yolov5 labels=/etc/labels/yolov5.labels \
constants="YoloV5,q-offsets=<3.0>,q-scales=<0.005047998391091824>;" ! video/x-
raw,format=BGRA,width=640,height=360 ! queue ! mixer.
```

To stop the use case, use **CTRL + C**.

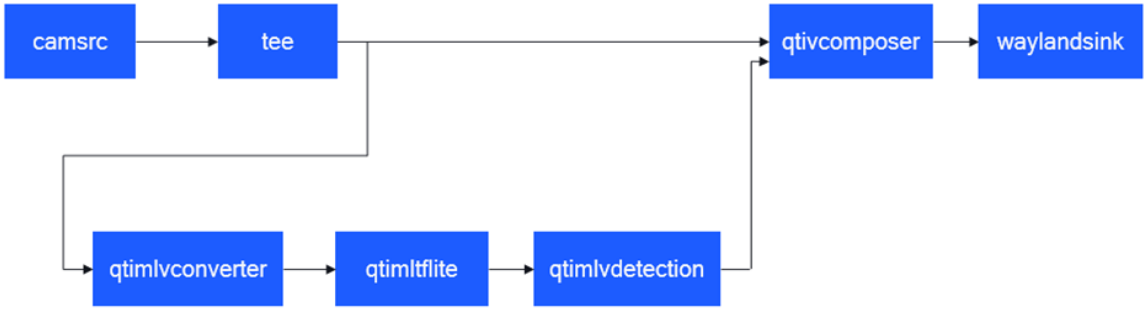


Figure 5-9 Pipeline for bounding box mask with qtivcomposer

The figure shows the flow of the use case execution:

1. Identifies object scenes in the scene from a video stream, which is coming through a camera source.
2. Composes the following using qtivcomposer:
   a. Bounding boxes over objects detected.
   b. Original video stream.
3. Displays the results.

The table provides the sequential processing stages of the pipeline execution:

Table 5-8 Pipeline processing stages for bounding box mask with qtivcomposer

Process	Description
qtiqmmfsrc	1. Collects the video stream (source) and creates two copies of the source: ▫ One stream is sent to qtivcomposer plugin to retain the video stream. ▫ The other stream is sent to a ML inferencing pipeline.
Preprocessing	
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs preprocessing: ▫ Color conversion ▫ Scaling down/up ▫ Normalization on the stream data when the model expects the floating point values as input 3. Converts the video stream to a tensor stream on its source pad. The object detection model uses this tensor stream for inferencing.
Inferencing	
qtimltflite	1. Loads the object detection model. 2. Modifies the graph for the chosen delegate. 3. Receives the tensor stream on its sinkpad. 4. Runs the inference and produces a tensor stream with the object detection results on its source pad.
Postprocessing	

Table 5-8 Pipeline processing stages for bounding box mask with qtivcomposer (cont.)

Process	Description
qtimlvdetection	1. Receives the inference tensors from the object detection model. 2. Converts the inference tensors on its sinkpad into formats like video or text that the multimedia plugins can process later. 3. Applies the threshold to the chosen number of results. 4. Loads the corresponding modules for detection models. In this use case, qtimlvdetection does the following: a. Loads the YOLOv5 submodule. b. Produces video frames with only bounding boxes that can be overlaid on objects. c. Sends them to the sinkpad of qtivcomposer.
qtivcomposer	1. Receives the original video stream and video stream with bounding boxes on its sinkpads. 2. On its sourcepads, produces content that's composed of the video streams processed from its sinkpads.
Output	
Waylandsink	1. Receives the video stream on its sinkpad. 2. Submits the video stream to Weston. 3. Weston displays the following on a local display device: a. The video stream is captured from the camera. b. The bounding boxes are drawn over the allowed number of objects identified in that scene.

5.1.1.5 Object detection and encode with LiteRT

The use cases use a YOLOv5 LiteRT model to identify the object in a scene. The use case is to either overlay or compose the bounding boxes over the detected objects, and then encode this stream as an H.264 bitstream.

Use qtivoverlay plugin to apply bounding box overlay

Run the use case:

```
gst-launch-1.0 -e qtiqmmfsrc name=camsrc ! video/x-
raw,format=NV12_Q08C,width=1280,height=720,framerate=30/1 ! queue ! tee
name=split split. ! \
queue ! qtimetamux name=metamux ! queue ! qtivoverlay ! queue ! v4l2h264enc
capture-io-mode=4 output-io-mode=5 ! h264parse ! queue ! mp4mux ! \
queue ! filesink location=/etc/media/video.mp4 split. ! queue !
qtimlvconverter ! queue ! qtimltflite delegate=external \
external-delegate-path=libQnnTFLiteDelegate.so external-delegate-
options="QNNExternalDelegate,backend_type=htp;" model=/etc/models/
yolov5.tflite ! queue ! \
qtimlvdetection threshold=75.0 results=10 module=yolov5 labels=/etc/labels/
yolov5.labels \
constants="YoloV5,q-offsets=<3.0>,q-scales=<0.005047998391091824>;" ! text/x-
raw ! queue ! metamux.
```

To stop the use case, use **CTRL + C**.

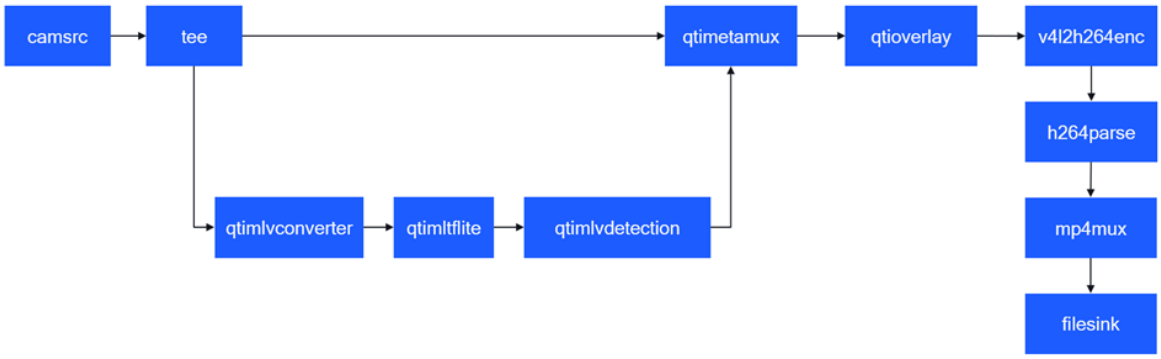


Figure 5-10 Pipeline for bounding box overlay and encode

The figure shows the flow of the use case execution:

1. Identify the object scenes from a video stream, which is coming through a camera source.
2. Overlay bounding boxes over the detected objects using overlaylib.
3. Encode the stream as a H.264 bitstream.
4. Multiplex the stream in an MP4 container and stored as an MP4 file.

The table provides the sequential processing stages of the pipeline execution:

Table 5-9 Pipeline processing stages for bounding box overlay and encode

Process	Description
qtiqmmfsrc	1. Collects the video stream (source) and creates two copies of the source: ▫ One stream is sent to qtimetamux plugin to retain the video stream. ▫ The other stream is sent to an ML inferencing pipeline.
Preprocessing	
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs preprocessing: ▫ Color conversion ▫ Scaling down/up ▫ Normalization on the stream data when the model expects the floating point values as an input 3. Converts the video stream to a tensor stream on its source pad. The object detection model uses this tensor stream for inferencing.
Inferencing	
qtimltflite	1. Loads the object detection model. 2. Modifies the graph for the chosen delegate. 3. Receives the tensor stream on its sinkpad. 4. Runs the inference and produces a tensor stream with the object detection results on its source pad.
Postprocessing	

Table 5-9 Pipeline processing stages for bounding box overlay and encode (cont.)

Process	Description
qtimlvdetection	1. Receives the inference tensors from the object detection model. 2. Converts the inference tensors on its sinkpad into formats like video or text that the multimedia plugins can process later. 3. Applies the threshold to the chosen number of results. 4. Loads the corresponding modules for detection models. In this use case, qtimlvdetection does the following: a. Loads the YOLOv5 submodule. b. Produces results as structures of text. c. Sends them to the sinkpad of qtimetamux.
qtimetamux	1. Receives video stream and text stream with bounding box results corresponding to the video stream on its sinkpads. 2. Produces GST buffers with contents of the video stream from its sink pad. 3. Adds bounding boxes as GstVideoRegionOfInterest from data sinkpad to GST buffers meta (meta muxing) on its source pad.
qtivoverlay	1. Receives the multiplexed stream. 2. Overlays the bounding boxes on the VideoFrame using CL. 3. Produces GST buffers with overlays in its source pad.
v4l2h264enc	1. Applies parameters to each frame of the video stream it's receiving on its sinkpad. 2. Encodes it into bitstream and sends it over its sourcepad.
h264parse	Adds more information about the bitstream to the GStreamer buffer meta.
mp4mux	Receives these buffers and creates containers with format specification buffers.
Output	
Filesink	Stores the resulting stream in a `/opt/video.mp4` file.
Playback	Pull `video.mp4` from the host computer and play it on a media player: `scp root@<IP address of target device>:/opt/ <destination directory>`

Use qtivcomposer to mix original frame with bounding box mask

Run the use case:

```
gst-launch-1.0 -e \
qtiqmmfsrc name=camsrc ! video/x-
raw,format=NV12_Q08C,width=1280,height=720,framerate=30/1 ! queue ! tee
name=split \
split. ! queue ! qtivcomposer name=mixer ! queue ! video/x-
raw,format=NV12,width=1920,height=1080,interlace-
mode=progressive,colorimetry=bt601 ! \
v4l2h264enc capture-io-mode=4 output-io-mode=5 ! h264parse ! queue ! mp4mux !
queue ! filesink location=/etc/media/video.mp4 \
split. ! queue ! qtimlvconverter ! queue ! qtimltflite delegate=external
external-delegate-path=libQnnTFLiteDelegate.so \
\external-delegate-options="QNNExternalDelegate,backend_type=htp;" model=/etc/
models/yolov5.tflite ! queue ! \
```

```
qtimlvdetection threshold=75.0 results=10 module=yolov5 labels=/etc/labels/
yolov5.labels \
constants="YoloV5,q-offsets=<3.0>,q-scales=<0.005047998391091824>;" ! video/x-
raw,format=BGRA,width=640,height=360 ! queue ! mixer.
```

To stop the use case, use **CTRL + C**.

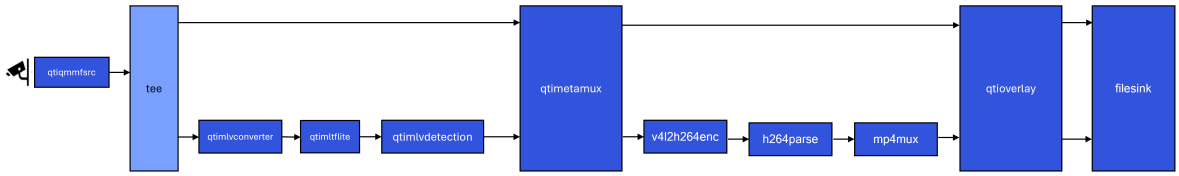


Figure 5-11 Pipeline for bounding box mask and encode with qtivcomposer

The figure shows the flow of the use case execution:

1. Identify object scenes from a video stream, which is coming through a camera source.
2. Using qtivcomposer, compose bounding boxes over the objects detected and the original video stream.
3. Encode this stream as an H.264 bitstream.
4. Multiplex the stream in an MP4 container and stored as an MP4 file.

The table provides the sequential processing stages of the pipeline execution:

Table 5-10 Pipeline processing stages for bounding box mask and encode with qtivcomposer

Process	Description
qtiqmmfsrc	1. Collects the video stream (source) and creates two copies of the source: ▫ One stream is sent to qtimetamux plugin to retain the video stream. ▫ The other stream is sent to an ML inferencing pipeline.
Preprocessing	
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs preprocessing: ▫ Color conversion ▫ Scaling down/up ▫ Normalization on the stream data when the model expects the floating point values as input 3. Converts the video stream to a tensor stream on its source pad. The object detection model uses this tensor stream for inferencing.
Inferencing	
qtimltflite	1. Loads the object detection model. 2. Modifies the graph for the chosen delegate. 3. Receives the tensor stream on its sinkpad. 4. Runs the inference and produces a tensor stream with the object detection results on its source pad.
Postprocessing	

Table 5-10 Pipeline processing stages for bounding box mask and encode with qtivcomposer (cont.)

Process	Description
qtimlvdetection	1. Receives the inference tensors from the object detection model. 2. Converts the inference tensors on its sinkpad into formats like video or text that the multimedia plugins can process later. 3. Applies the threshold to the chosen number of results. 4. Loads the corresponding modules for detection models. In this use case, qtimlvdetection does the following: a. Loads the YOLOv5 submodule. b. Produces video frames with only bounding boxes that can be overlaid on objects. c. Sends them to the sinkpad of qtivcomposer.
qtivcomposer	1. Receives the original video stream and video stream with bounding boxes on its sinkpads 2. On its sourcepads, produces content that's composed of video streams processed from its sinkpads.
v4l2h264enc	1. Applies parameters to each frame of the video stream it's receiving on its sinkpad. 2. Encodes it into bitstream and sends it over its sourcepad.
h264parse	Adds more information about the bitstream to the GStreamer buffer meta.
mp4mux	Receives these buffers and creates containers with format specification buffers.
Output	
Filesink	Stores the resulting stream in a `/etc/media/video.mp4` file.
Playback	Pull `video.mp4` from the host computer and play it on a media player: `scp root@<IP address of target device>:/etc/ <destination directory>`

5.1.1.6 Image segmentation and display with LiteRT

The use case uses the `deeplabv3_resnet50` LiteRT model to identify semantic segmentations in a scene from a camera stream. The use case is to compose the semantics and original video stream using qtivcomposer, and then display the results.

Run the use case:

```
gst-launch-1.0 -e --gst-debug=2 \
qtiqmmfsrc name=camsrc ! video/x-
raw,format=NV12_Q08C,width=1280,height=720,framerate=30/1 ! queue ! tee
name=split \
split. ! queue ! qtivcomposer name=mixer sink_1::alpha=0.5 ! queue !
waylandsink fullscreen=true sync=false \
split. ! queue ! qtimlvconverter ! queue ! qtimltflite delegate=external
external-delegate-path=libQnnTFLiteDelegate.so \
external-delegate-options="QNNExternalDelegate,backend_type=htp;" model=/etc/
models/deeplabv3_plus_mobilenet_quantized.tflite ! queue ! \
qtimlvsegmentation module=deeplab-argmax labels=/etc/labels/
deeplabv3_resnet50.labels constants="deeplab,q-offsets=<0.0>,q-
```

```
scales=<1.0>;" ! \
video/x-raw,width=256,height=144 ! queue ! mixer.
```

To stop the use case, use **CTRL + C**.

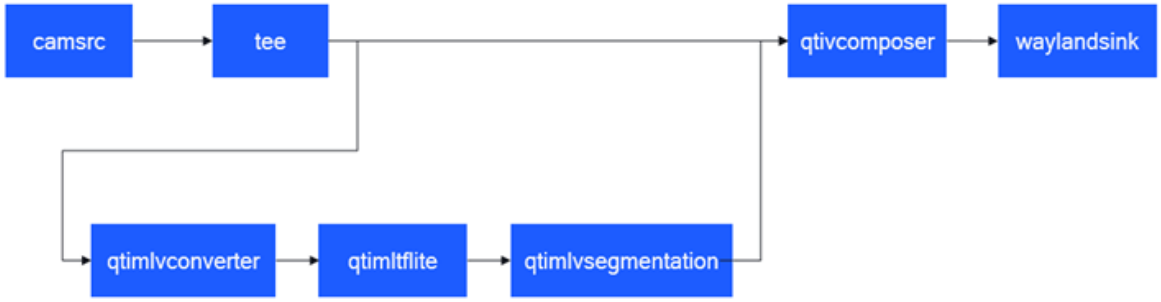


Figure 5-12 Pipeline for segmentation with qtivcomposer

The figure shows the flow of the use case execution:

1. Identify scenes from a video stream coming through a camera source.
2. Compose semantic segmentation and video stream using qtivcomposer.
3. Display the results.

The table provides the sequential processing stages of the pipeline execution:

Table 5-11 Pipeline processing stages for segmentation with qtivcomposer

Process	Description
qtiqmmfsrc	1. Collects the video stream (source) and creates two copies of the source: ▫ One stream is sent to the qtivcomposer plugin to retain the video stream. ▫ The other stream is sent to the ML inferencing branch in the pipeline.
Preprocessing	
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs preprocessing: ▫ Color conversion ▫ Scaling down/up ▫ Normalization on the stream data when the model expects the floating point values as input 3. Converts the video stream to a tensor stream on its source pad. The segmentation model uses this tensor stream for inferencing.
Inferencing	
qtimltflite	1. Loads the segmentation model. 2. Modifies the graph for the chosen delegate. 3. Receives the tensor stream on its sinkpad. 4. Runs the inference and produces a tensor stream with the segmentation results on its source pad.
Postprocessing	

Table 5-11 Pipeline processing stages for segmentation with qtivcomposer (cont.)

Process	Description
qtimlvsegmentation	1. Receives the inference tensors on its sinkpad. 2. Converts the inference tensors into video formats that the multimedia plugins can process later. 3. Produces the semantic segmentations for the frame. 4. Loads the corresponding modules for the segmentation models. In this use case, qtimlvsegmentation does the following: a. Loads the deeplab-argmax submodule. b. Produces video frames with segmentation masks. c. Sends them to the sinkpad of qtivcomposer.
qtivcomposer	1. Receives the original video stream with segmentation mask on its sinkpads. 2. On its sourcepad, produces GST buffers with contents composed of video streams from its sinkpads.
Output	
Waylandsink	1. Receives the video stream on its sinkpad. 2. Submits the video stream to Weston. 3. Weston displays the following on the local display device: ▫ The video stream that's captured from the camera. ▫ The segmentation masks that are drawn over objects/components in that scene.

5.1.1.7 Image segmentation and encode with LiteRT

The use case uses the `deeplabv3_resnet50` LiteRT model to compose the semantic segmentations and original video stream, encode this stream, and then multiplex it in an MP4 container.

Run the use case:

```
gst-launch-1.0 -e --gst-debug=2 \
qtiqmmfsrc name=camsrc ! video/x-
raw,format=NV12_Q08C,width=1280,height=720,framerate=30/1 ! queue ! tee
name=split \
split. ! queue ! qtivcomposer name=mixer sink_1::dimensions="<1920,1080>"
sink_1::alpha=0.5 ! queue ! video/x-
raw,format=NV12,width=1920,height=1080,interlace-
mode=progressive,colorimetry=bt601 ! \
v4l2h264enc capture-io-mode=4 output-io-mode=5 ! h264parse ! queue ! mp4mux !
queue ! filesink location=/etc/media/video.mp4 \
split. ! queue ! qtimlvconverter ! queue ! qtimltflite delegate=external
external-delegate-path=libQnnTFLiteDelegate.so \
external-delegate-options="QNNExternalDelegate,backend_type=htp;" model=/etc/
models/deeplabv3_plus_mobilenet_quantized.tflite ! queue ! \
qtimlvsegmentation module=deeplab-argmax labels=/etc/labels/
deeplabv3_resnet50.labels constants="deeplab,q-offsets=<0.0>,q-
scales=<1.0>;" ! video/x-raw,width=256,height=144 ! queue ! mixer.
```

To stop the use case, use **CTRL + C**.

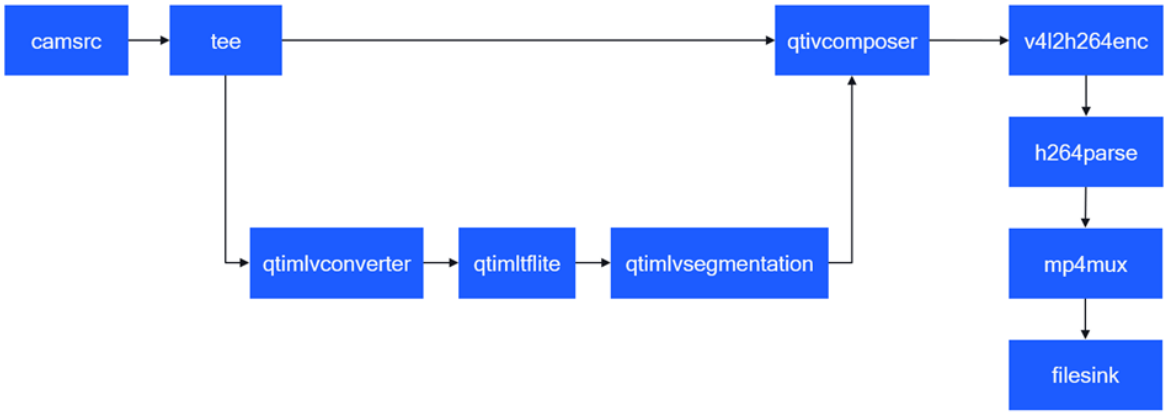


Figure 5-13 Pipeline for segmentation and encode with qtivcomposer

The figure shows the flow of the use case execution:

1. Identify the scenes from a video stream coming through a camera source.
2. Compose the semantic segmentation and video stream using qtivcomposer.
3. Encode the stream as a H.264 bit stream and multiplex the stream in an MP4 container.

The table provides the sequential processing stages of the pipeline execution:

Table 5-12 Pipeline processing stages for segmentation and encode with qtivcomposer

Process	Description
qtiqmmfsrc	1. Collects the video stream (source) and creates two copies of the source: ▫ One stream is sent to the qtivcomposer plugin to retain the video stream. ▫ The other stream is sent to an ML inferencing branch in the pipeline.
Preprocessing	
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs preprocessing: ▫ Color conversion ▫ Scaling down/up ▫ Normalization on the stream data when the model expects the floating point values as an input 3. Converts the video stream to a tensor stream on its source pad. The segmentation model uses this tensor stream for inferencing.
Inferencing	
qtimltflite	1. Loads the segmentation model. 2. Modifies the graph for the chosen delegate. 3. Receives the tensor stream on its sinkpad. 4. Runs the inference and produces a tensor stream with the segmentation results on its source pad.
Postprocessing	

Table 5-12 Pipeline processing stages for segmentation and encode with qtivcomposer (cont.)

Process	Description
qtimlvsegmentation	1. Receives the inference tensors on its sinkpad. 2. Converts the inference tensors into video formats that the multimedia plugins can process later. 3. Produces the semantic segmentations for the frame. 4. Loads the corresponding modules for the segmentation models. In this use case, qtimlvsegmentation does the following: a. Loads the deeplab-argmax submodule. b. Produces video frames with segmentation masks. c. Sends them to the sinkpad of qtivcomposer.
qtivcomposer	1. Receives the original video stream with segmentation mask on its sinkpads. 2. Produces on its sourcepad GST buffers with contents composed of video streams from its sinkpads.
v4l2h264enc	1. Applies parameters to each frame of the video stream it's receiving on its sinkpad. 2. Encodes it into bit stream and sends it over its sourcepad.
h264parse	Adds more information about the bit stream to the GStreamer buffer meta.
mp4mux	Receives these buffers and creates the container format specification buffers.
Output	
Filesink	Stores the resulting stream in a `/etc/media/video.mp4` file.
Playback	Pull `video.mp4` from the host computer and play it on a media player: `scp root@<IP address of target device>:/etc/ <destination directory>`

5.1.1.8 Pose estimation and display with LiteRT

The use cases use the PoseNet LiteRT model to process a single camera stream with pose estimation.

Use qtivoverlay plugin to apply pose estimation overlay

Run the use case:

```
gst-launch-1.0 -e \
qtiqmmfsrc name=camsrc ! video/x-
raw,format=NV12_Q08C,width=1280,height=720,framerate=30/1 ! queue ! tee
name=split \
split. ! queue ! qtimetamux name=metamux ! queue ! qtivoverlay ! queue !
waylandsink fullscreen=true sync=false \
split. ! queue ! qtimlvconverter ! queue ! qtimltflite delegate=external
external-delegate-path=libQnnTFLiteDelegate.so \
external-delegate-options="QNNExternalDelegate,backend_type=htp;" model=/etc/
models/hrnet_pose_quantized.tflite ! queue ! \
qtimlvpose threshold=51.0 results=2 module=hrnet labels=/etc/labels/
hrnet_pose.labels \
```

```
constants="hrnet,q-offsets=<8.0>,q-scales=<0.0040499246679246426>;" ! text/x-raw ! queue ! metamux.
```

To stop the use case, use **CTRL + C**.

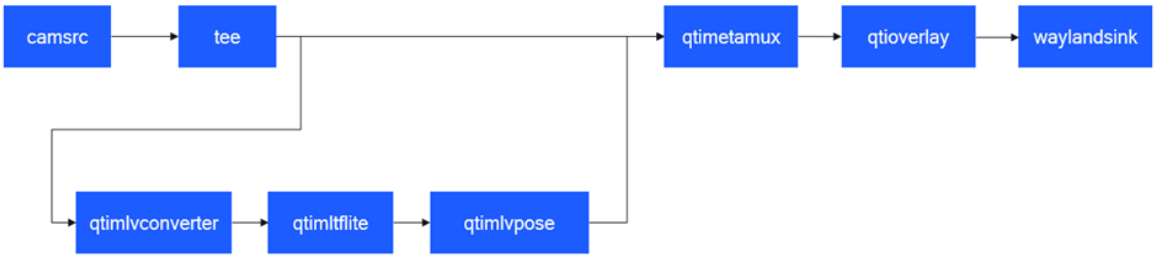


Figure 5-14 Pipeline for pose estimation overlay

The figure shows the flow of the use case execution:

1. Identify poses of people in scenes from a video stream, which is coming through a camera source.
2. Overlay the available poses using overlaylib.
3. Display the results.

The table provides the sequential processing stages of the pipeline execution:

Table 5-13 Pipeline processing stages for pose estimation overlay

Process	Description
qtiqmmfsrc	1. Collects the video stream (source) and creates two copies of the source: a. One is sent to the qtimetamux plugin to retain the video stream. b. The other is sent to an ML inferencing pipeline.
Preprocessing	
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs preprocessing: ▫ Color conversion ▫ Scaling down/up ▫ Normalization on the stream data when the model expects the floating point values as input 3. Converts the video stream to a tensor stream on its source pad. The PoseNet model uses this tensor stream for inferencing.
Inferencing	
qtimltflite	1. Loads the PoseNet model. 2. Modifies the graph for the chosen delegate. 3. Receives the tensor stream on its sinkpad. 4. Runs the inference and produces a tensor stream with the pose estimation results on its source pad.
Postprocessing	
qtimlvpose	1. Receives the inference tensors from a PoseNet model on its sinkpad. 2. Converts the tensors into formats such as video or text that the multimedia plugins can process later.

Table 5-13 Pipeline processing stages for pose estimation overlay (cont.)

Process	Description
	3. Applies the threshold to the chosen number of results. 4. Loads the corresponding modules of the pose estimation models. In this use case, qtimlvpose does the following: a. Loads the PoseNet submodule. b. Produces results as structures of text. c. Sends them to the sinkpad of qtimetamux.
qtimetamux	1. Receives the video and text streams with pose results corresponding to the video stream on its sinkpads. 2. Produces GST buffers with the contents of video stream on its sink pad. 3. Adds poses from data sinkpad to GST buffer meta (meta muxing) on its source pad.
qtivoverlay	1. Receives the multiplexed stream. 2. Overlays the poses on the VideoFrame using CL. 3. Produces GST buffers with overlays in its source pad.
Output	
Waylandsink	1. Receives the video stream on its sinkpad. 2. Submits the video stream to Weston. 3. Weston renders the following on a local display device: a. The video stream captured from the camera. b. The poses generated for several people in that scene.

Use qtivcomposer to mix original frame with pose estimation mask

Run the use case:

```
gst-launch-1.0 -e --gst-debug=2 \
qtiqmmfsrc name=camsrc ! video/x-
raw,format=NV12_Q08C,width=1280,height=720,framerate=30/1 ! queue ! tee
name=split \
split. ! queue ! qtivcomposer name=mixer ! queue ! waylandsink
fullscreen=true sync=false \
split. ! queue ! qtimlvconverter ! queue ! qtimltflite delegate=external
external-delegate-path=libQnnTFLiteDelegate.so external-delegate-
options="QNNExternalDelegate,backend_type=htp;" \
model=/etc/models/hrnet_pose_quantized.tflite ! queue ! qtimlvpose
threshold=51.0 results=2 module=hrnet labels=/etc/labels/hrnet_pose.labels \
constants="hrnet,q-offsets=<8.0>,q-scales=<0.0040499246679246426>;" ! video/x-
raw,format=BGRA,width=640,height=360 ! queue ! mixer.
```

To stop the use case, use **CTRL + C**.

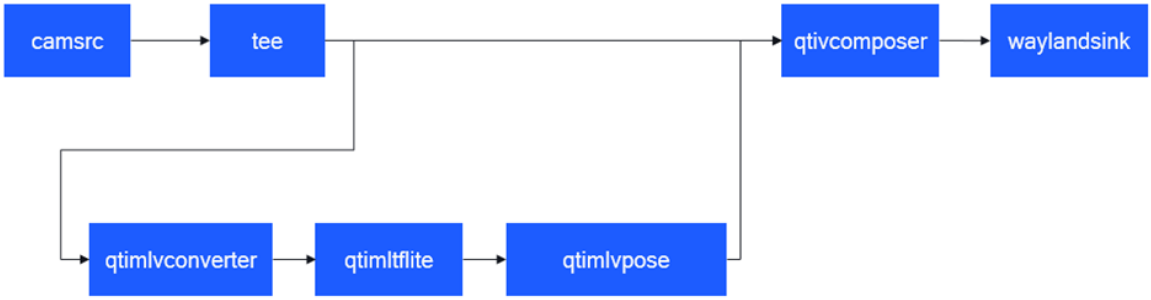


Figure 5-15 Pipeline for pose estimation mask using qtivcomposer

The figure shows the flow of the use case execution:

1. Identify poses of people in the scenes from a video stream, which is coming through a camera source.
2. Compose the poses and video stream using qtivcomposer.
3. Display the results.

The table provides the sequential processing stages of the pipeline execution:

Table 5-14 Pipeline processing stages for pose estimation mask using qtivcomposer

Process	Description
qtiqmmfsrc	1. Collects the video stream (source) and creates two copies of the source: ▫ One is sent to the qtimetamux plugin to retain the video stream. ▫ The other is sent to an ML inferencing pipeline.
Preprocessing	
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs preprocessing: ▫ Color conversion ▫ Scaling down/up ▫ Normalization on the stream data when the model expects the floating point values as input 3. Converts the video stream to a tensor stream on its source pad. The PoseNet model uses this tensor stream for inferencing.
Inferencing	
qtimltflite	1. Loads the PoseNet model. 2. Modifies the graph for the chosen delegate. 3. Receives the tensor stream on its sinkpad. 4. Runs the inference and produces a tensor stream with the pose estimation results on its source pad.
Postprocessing	
qtimlvpose	1. Receives the inference tensors from a PoseNet model on its sinkpad. 2. Converts the tensors into formats such as video or text that the multimedia plugins can process later.

Table 5-14 Pipeline processing stages for pose estimation mask using qtivcomposer (cont.)

Process	Description
	3. Applies the threshold to the chosen number of results. 4. Loads the corresponding modules of the pose estimation models. In this use case, qtimlvpose does the following: a. Loads the PoseNet submodule. b. Produces results as video frames with poses drawn. c. Sends them to the sinkpad of qtivcomposer.
qtivcomposer	1. Receives the original video stream and the video stream of poses on its sinkpads. 2. On its sourcepad, produces the GST buffers with the contents composed of video streams from its sinkpads.
Output	
Waylandsink	1. Receives the video stream on its sinkpad. 2. Submits the video stream to Weston. 3. Weston renders the following on a local display device: a. The video stream captured from the camera. b. The poses generated for several people in that scene.

5.1.1.9 Pose estimation and encode with LiteRT

The use cases use the PoseNet LiteRT model to process a single camera stream with pose estimation and encode the stream as an H.264 bitstream.

Use qtivoverlay plugin to apply pose estimation overlay

Run the use case:

```
gst-launch-1.0 -e \
qtiqmmfsrc name=camsrc ! video/x-
raw,format=NV12_Q08C,width=1280,height=720,framerate=30/1 ! queue ! tee
name=split \
split. ! queue ! qtimetamux name=metamux ! queue ! qtivoverlay ! queue !
v4l2h264enc capture-io-mode=4 output-io-mode=5 ! \
h264parse ! queue ! mp4mux ! queue ! filesink location=/etc/media/video.mp4 \
split. ! queue ! qtimlvconverter ! queue ! qtimltflite delegate=external
external-delegate-path=libQnnTFLiteDelegate.so \
external-delegate-options="QNNExternalDelegate,backend_type=htp;" model=/etc/
models/hrnet_pose_quantized.tflite ! queue ! \
qtimlvpose threshold=51.0 results=2 module=hrnet labels=/etc/labels/
hrnet_pose.labels \
constants="hrnet,q-offsets=<8.0>,q-scales=<0.0040499246679246426>;" ! text/x-
raw ! queue ! metamux.
```

To stop the use case, use **CTRL + C**.

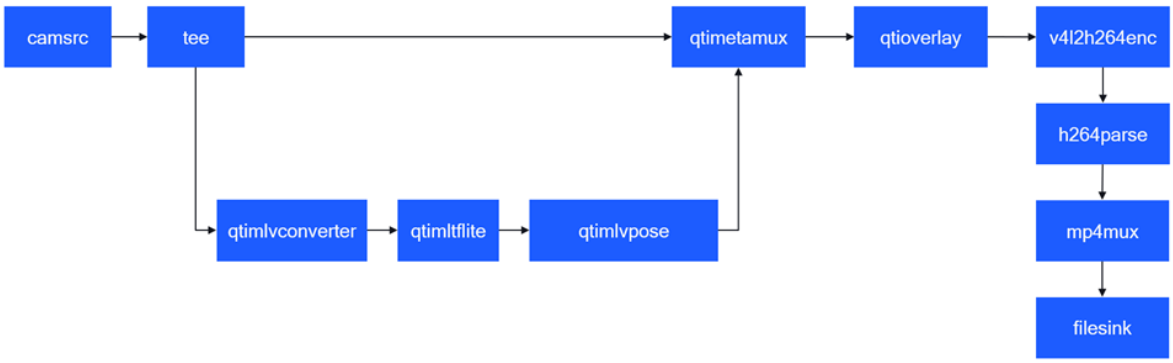


Figure 5-16 Pipeline for pose estimation and encode using qtioverlay

The figure shows the flow of the use case execution:

1. Identify poses of people in the scenes from the video stream coming through the camera source.
2. Overlay the available poses using overlaylib.
3. Encode the stream as an H.264 bitstream.
4. Multiplex the stream in an MP4 container and store it as an MP4 file.

The table provides the sequential processing stages of the pipeline execution:

Table 5-15 Pipeline processing stages for pose estimation and encode using qtioverlay

Process	Description
qtiqmmfsrc	1. Collects the video stream (source) and creates two copies of the source: ▫ One stream is sent to the qtimetamux plugin to retain the video stream. ▫ The other stream is sent to an ML inferencing pipeline.
Preprocessing	
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs preprocessing: ▫ Color conversion ▫ Scaling down/up ▫ Normalization on the stream data when the model expects the floating point values as input 3. Converts the video stream to a tensor stream on its source pad. The PoseNet model uses this tensor stream for inferencing.
Inferencing	
qtimltflite	1. Loads the PoseNet model. 2. Modifies the graph for the chosen delegate. 3. Receives the tensor stream on its sinkpad. 4. Runs the inference and produces a tensor stream with the pose estimation results on its source pad.
Postprocessing	

Table 5-15 Pipeline processing stages for pose estimation and encode using qtioverlay (cont.)

Process	Description
qtimlvpose	1. Receives the inference tensors from a PoseNet model on its sinkpad. 2. Converts the tensors into formats such as video or text that the multimedia plugins can process later. 3. Applies the threshold to the chosen number of results. 4. Loads the corresponding modules of the pose estimation models. In this use case, qtimlvpose does the following: a. Loads the PoseNet submodule. b. Produces results as structures of text. c. Sends them to the sinkpad of qtimetamux.
qtimetamux	1. Receives the video stream and text stream with pose results corresponding to video stream on its sinkpads. 2. Produces GST buffers with the contents of video stream on its sink pad. 3. Adds poses from data sinkpad to GST buffer meta (meta muxing) on its source pad.
qtivoverlay	1. Receives the multiplexed stream. 2. Overlays the poses on the VideoFrame using CL. 3. Produces GST buffers with overlays in its source pad.
v4l2h264enc	1. Applies parameters to each frame of the video stream it's receiving on its sinkpad. 2. Encodes it into bitstream and sends it over its sourcepad.
h264parse	Adds more information about the bitstream to the GStreamer buffer meta.
mp4mux	Receives the buffers and creates containers format specification buffers.
Output	
Filesink	Stores the resulting stream in a `/opt/video.mp4` file.
Playback	Pull `video.mp4` from the host computer and play it on a media player: `scp root@<IP address of target device>:/opt/ <destination directory>`

Use qtivcomposer to mix original frame with pose estimation mask

Run the use case:

```
gst-launch-1.0 -e --gst-debug=2 \
qtiqmmfsrc name=camsrc ! video/x-
raw,format=NV12_Q08C,width=1280,height=720,framerate=30/1 ! queue ! tee
name=split \
split. ! queue ! qtivcomposer name=mixer sink_1::dimensions="<1920,1080>" !
queue ! video/x-raw,format=NV12,width=1920,height=1080,interlace-
mode=progressive,colorimetry=bt601 ! \
v4l2h264enc capture-io-mode=4 output-io-mode=5 ! h264parse ! queue ! mp4mux !
queue ! filesink location=/etc/media/video.mp4 \
split. ! queue ! qtimlvconverter ! queue ! qtimltflite delegate=external
external-delegate-path=libQnnTFLiteDelegate.so external-delegate-
options="QNNExternalDelegate,backend_type=htp;" \
```

```
model=/etc/models/hrnet_pose_quantized.tflite ! queue ! qtimlvpose
threshold=51.0 results=2 module=hrnet labels=/etc/labels/hrnet_pose.labels \
constants="hrnet,q-offsets=<8.0>,q-scales=<0.0040499246679246426>;" ! video/x-
raw,format=BGRA,width=640,height=360 ! queue ! mixer.
```

To stop the use case, use **CTRL + C**.

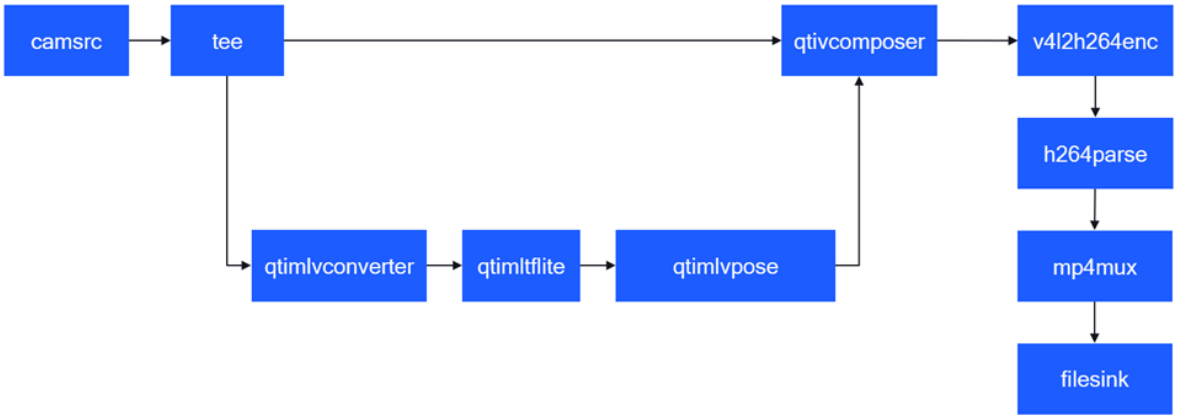


Figure 5-17 Pipeline for pose estimation and encode using qtivcomposer

The figure shows the flow of the use case execution:

1. Classify scenes from the video stream coming through a camera source.
2. Compose the poses and video stream together using qtivcomposer.
3. Encode this stream as an H.264 bitstream.
4. Multiplex in an MP4 container and storing it as an MP4 file.

The table provides the sequential processing stages of the pipeline execution:

Table 5-16 Pipeline processing stages for pose estimation and encode using qtivcomposer

Process	Description
qtiqmmfsrc	1. Collects the video stream (source) and creates two copies of the source: ▫ One stream is sent to the qtivcomposer plugin to retain the video stream. ▫ The other stream is sent to an ML inferencing pipeline.
Preprocessing	
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs preprocessing: ▫ Color conversion ▫ Scaling down/up ▫ Normalization on the stream data when the model expects the floating point values as input 3. Converts the video stream to a tensor stream on its source pad. The PoseNet model uses this tensor stream for inferencing.
Inferencing	

Table 5-16 Pipeline processing stages for pose estimation and encode using qtivcomposer (cont.)

Process	Description
qtimltflite	1. Loads the PoseNet model. 2. Modifies the graph for the chosen delegate. 3. Receives the tensor stream on its sinkpad. 4. Runs the inference and produces a tensor stream with the pose estimation results on its source pad.
Postprocessing	
qtimlvpose	1. Receives the inference tensors from a PoseNet model on its sinkpad. 2. Converts the tensors into formats such as video or text that the multimedia plugins can process later. 3. Applies the threshold to the chosen number of results. 4. Loads the corresponding modules of the pose estimation models. In this use case, qtimlvpose does the following: a. Loads the PoseNet submodule. b. Produces results as video frames with poses drawn. c. Sends them to the sinkpad of the qtivcomposer.
	1. Receives the original video stream and video stream of poses on its sinkpads. 2. On its sourcepad, produces the GST buffers with the contents composed of video streams from its sinkpads.
qtivoverlay	1. Receives the multiplexed stream. 2. Overlays the poses on the VideoFrame using CL. 3. Produces GST buffers with overlays in its source pad.
v4l2h264enc	1. Applies parameters to each frame of the video stream that it's receiving on its sinkpad. 2. Encodes it into bitstream and sends it over its sourcepad.
h264parse	Adds more information about the bit stream to the GStreamer buffer meta.
mp4mux	Receives the buffers and creates containers format specification buffers.
Output	
Filesink	Stores the resulting stream in a `/etc/media/video.mp4` file.
Playback	Pull `video.mp4` from the host computer and play it on a media player: `scp root@<IP address of target device>:/etc/ <destination directory>`

5.1.1.10 Video super resolution and display with LiteRT

Video super resolution (VSR) is supported on AI Hub quantized INT8 models with 128 ×128 input resolution and 512 × 512 output resolution.

Select QuickSRNetSmall-Quantized to download the model file.

Run the use case with file source (128 ×128 input resolution):

```
gst-launch-1.0 -e --gst-debug=2 \
filesrc location=/etc/media/video.mp4 ! qtdemux ! queue ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,format=NV12 ! \
```

```
queue !  tee name=split split. ! queue ! qtivcomposer name=mixer
sink_0::position="<0, 0>" sink_0::dimensions="<960, 1080>" \
sink_1::position="<960, 0>" sink_1::dimensions="<960, 1080>" ! queue !
waylandsink sync=true fullscreen=true split. ! \
qtimlvconverter   ! queue ! qtimltflite delegate=external external-delegate-
path=libQnnTFLiteDelegate.so \
external-delegate-options="QNNExternalDelegate,backend_type=htp;" model=/etc/
models/quicksrnetsmall_quantized.tflite ! \
queue ! qtimlvsuperresolution module=srnet constants="qsrnetsmall,q-
offsets=<0.0>,q-scales=<1.0>;" !  video/x-raw,format=RGB ! queue ! mixer.
```

The output is displayed on a Wayland display.

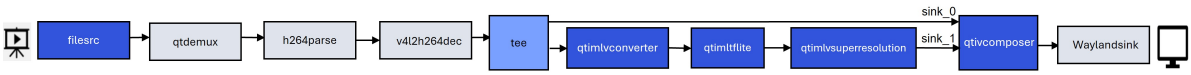


Figure 5-18 Pipeline for video resolution (Wayland display)

For information about expected output and plugins used in this pipeline, see the Video super-resolution sample application.

RELATED INFORMATION

"qtimlvsuperresolution" on page 419

5.1.1.11 Single stream from camera to RTSP with ML detection

Play a stream from the camera through RTSP on a media player (such as VLC).

- Single stream from camera to the RTSP:

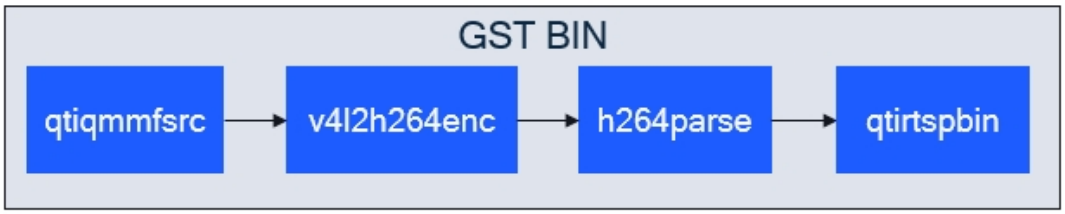


Figure 5-19 Pipeline for camera to RTSP

```
gst-launch-1.0 -e qtiqmmfsrc ! video/x-
raw,format=NV12_Q08C,width=1920,height=1080,framerate=30/1 ! \
v4l2h264enc capture-io-mode=4 output-io-mode=5 ! queue ! h264parse config-
interval=1 ! queue ! qtirtspbin
```

Connect to the stream. For more information, see View RTSP stream on host computer.

- Single stream from camera to the RTSP with ML detection meta stream:

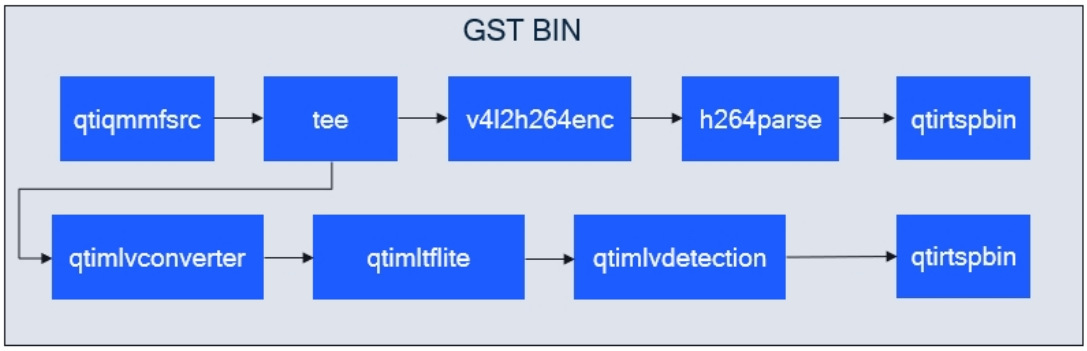


Figure 5-20 Pipeline for camera to RTSP with ML detection

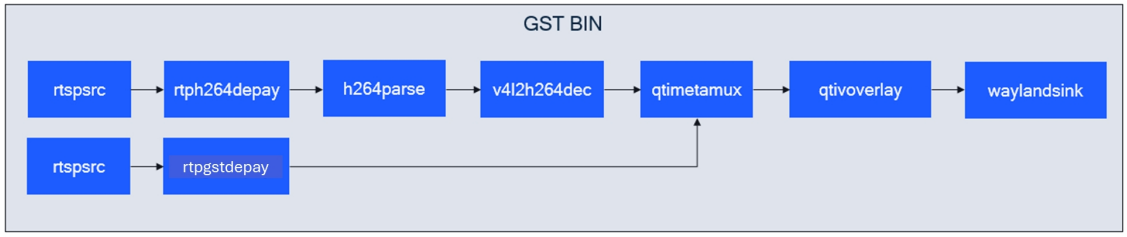


Figure 5-21 Preview with RTSP source

In console 1:

```
gst-launch-1.0 -e qtiqmmfsrc ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1 ! queue ! tee
name=split ! \
v4l2h264enc capture-io-mode=4 output-io-mode=5 ! queue ! h264parse config-
interval=-1 ! queue ! qtirtspbin name=rtsp split. ! queue ! \
qtimlvconverter ! queue ! qtimltflite delegate=external external-delegate-
path=libQnnTFLiteDelegate.so external-delegate-
options="QNNExternalDelegate,backend_type=htp;" \
model=/etc/models/yolov8_det_quantized.tflite ! queue ! qtimlvdetection
threshold=70.0 results=5 module=yolov8 labels=/etc/labels/yolov8.labels \
constants="yolov8,q-offsets=<21.0, 0.0, 0.0>,q-scales=<3.0546178817749023,
0.003793874057009816, 1.0>;" ! text/x-raw ! queue ! rtsp.
```

In console 2:

```
gst-launch-1.0 -e rtspsrc location=rtsp://<ip-address.of-device>:8900/
live ! rtph264depay ! video/x-h264,colorimetry=bt709 ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! queue ! qtimetamux
name=metamux ! queue ! qtivoverlay ! waylandsink fullscreen=true
async=true sync=false rtspsrc location=rtsp://<ip-address-of-device>:8900/
live ! rtpgstdepay ! text/x-raw ! queue ! metamux.
```

RELATED INFORMATION

"qtirtspbin" on page 427

5.1.2 Qualcomm Neural Processing SDK use cases

Qualcomm Neural Processing SDK (formerly known as Qualcomm Snapdragon Neural Processing Engine (SNPE)) is used to run deep neural networks for inference. The use cases describe the image classification, object detection, and image segmentation scenarios using different ML models.

NOTE The deep learning container (DLC) models used in the pipelines are available with the Qualcomm Neural Processing SDK release.

Before you run the use cases, complete the preconditions mentioned in GStreamer command-line use cases.

RELATED INFORMATION

"Configure Qualcomm GStreamer plugins" on page 343

5.1.2.1 Image classification and display with Neural Processing SDK

The use cases use an Inceptionv3 model with Qualcomm Neural Processing SDK to classify scenes, either overlay or compose the classification labels, and then display the results.

You can use any publicly available classification model with TensorFlow and convert it to the `.dlc` format as described in TensorFlow Model Conversion.

Use qtivoverlay plugin to apply classification overlay

Run the use case:

```
gst-launch-1.0 -e --gst-debug=2 \
qtiqmmfsrc name=camsrc ! video/x-
raw,format=NV12_Q08C,width=1280,height=720,framerate=30/1 ! queue ! tee
name=split \
split. ! queue ! qtimetamux name=metamux ! queue ! qtivoverlay ! queue !
waylandsink fullscreen=true sync=false \
split. ! queue ! qtimlvconverter ! queue ! qtimlsnpe delegate=dsp model=/etc/
models/inceptionv3.dlc ! queue ! \
qtimlvclassification threshold=40.0 results=2 module=mobilenet labels=/etc/
labels/classification.labels ! text/x-raw ! queue ! metamux.
```

To stop the use case, use **CTRL + C**.

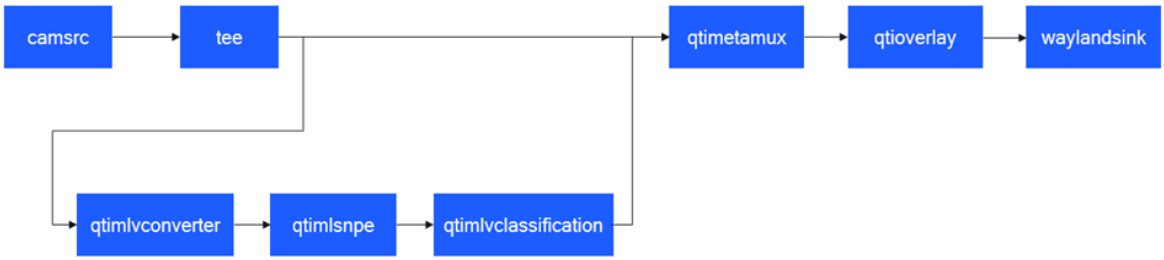


Figure 5-22 Pipeline for classification overlay

The figure shows the flow of the use case execution:

- Classify scenes from a video stream coming through a camera source.
- Overlay classification labels using overlaylib.
- Display the results on a local display.

The table provides the sequential processing stages of the pipeline execution:

Table 5-17 Pipeline processing stages for classification overlay

Process	Description
qtiqmmfsrc	1. Collects the video stream (source) and creates two copies of the source: ▫ One stream is sent to qtimetamux plugin to retain the video stream. ▫ The other stream is sent to an ML inferencing pipeline.
Preprocessing	
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs preprocessing: ▫ Color conversion ▫ Scaling down/up ▫ Normalization on the stream data when the model expects the floating point values as input 3. Converts the video stream to a tensor stream on its source pad. The classification model uses this tensor stream for inferencing.
Inferencing	
qtimlsnpe	1. Loads the model. 2. Modifies the graph for the chosen delegate. 3. Receives the tensor stream on its sinkpad. 4. Runs the inference and produces a tensor stream with the inference results on its source pad.
Postprocessing	
qtimlvclassification	1. Receives the inference tensors from the model on its sinkpad. 2. Converts the tensors into formats such as video or text that the multimedia plugins can process later. 3. Applies the threshold to the chosen number of results. 4. Loads the corresponding modules of the classification models. In this use case, qtimlvclassification does the following: a. Loads the submodule of the model. b. Produces results as structures of text. c. Sends them to the sinkpad of qtimetamux.
qtimetamux	1. Receives the video stream and text stream with classification results corresponding to the video stream on its sinkpads. 2. Produces GST buffers with the contents of video stream on its sink pad. 3. Adds classification result from data sinkpad to GST buffer meta (meta muxing) on its source pad.
qtivoverlay	1. Receives the multiplexed stream. 2. Overlays the classification labels on the VideoFrame using CL. 3. Produces GST buffers with overlays in its source pad.

Table 5-17 Pipeline processing stages for classification overlay (cont.)

Process	Description
Output	
Waylandsink	1. Receives the video stream on its sinkpad. 2. Submits the video stream to Weston. 3. Weston renders the video stream and possible classifications generated for that scene on a local display device.

Use qtivcomposer to mix original frame with classification mask

Run the use case:

```
gst-launch-1.0 -e --gst-debug=2 qtiqmmfsrc name=camsrc ! video/x-
raw,format=NV12_Q08C,width=1280,height=720,framerate=30/1 ! queue ! \
tee name=split split. ! queue ! qtivcomposer name=mixer
sink_1::position="<30, 30>" sink_1::dimensions="<320, 320>" ! queue !
waylandsink fullscreen=true \
split. ! queue ! qtimlvconverter ! queue ! qtimlsnpe delegate=dsp model=/etc/
models/inceptionv3.dlc ! queue ! \
qtimlvclassification threshold=40.0 results=2 module=mobilenet labels=/etc/
labels/classification.labels ! video/x-raw,format=BGRA,width=640,height=360 !
queue ! mixer.
```

To stop the use case, use **CTRL + C**.

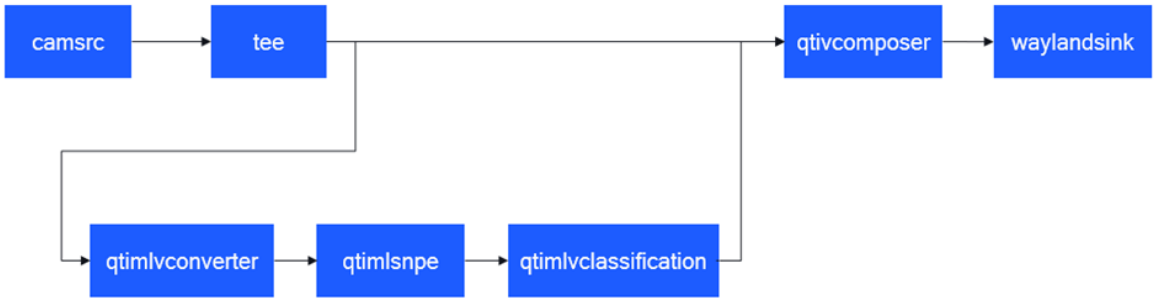


Figure 5-23 Pipeline for classification using qtivcomposer

The figure shows the flow of the use case execution:

- Classify scenes from a video stream coming through a camera source.
- Compose classification labels and video stream together using qtivcomposer.
- Display the results to a local display.

The table provides the sequential processing stages of the pipeline execution:

Table 5-18 Pipeline processing stages for classification using qtivcomposer

Process	Description
qtiqmmfsrc	1. Collects the video stream (source) and creates two copies of the source: ▫ One stream is sent to the qtivcomposer plugin to retain the video stream. ▫ The other stream is sent to the ML inferencing branch in the pipeline.
Preprocessing	
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs preprocessing: ▫ Color conversion ▫ Scaling down/up ▫ Normalization on the stream data when a model expects the floating point values as input 3. Converts the video stream to a tensor stream on its source pad. The classification model uses this tensor stream for inferencing.
Inferencing	
qtimlsnpe	1. Loads the model. 2. Modifies the graph for the chosen delegate. 3. Receives the tensor stream on its sinkpad. 4. Runs the inference and produces a tensor stream with the inference results on its source pad.
Postprocessing	
qtimlvclassification	1. Receives the inference results from the model on its sinkpad. 2. Converts the inference tensors into formats like video or text that the multimedia plugins can process later. 3. Applies the threshold to the chosen number of results. 4. Loads the corresponding modules for the classification models. In this use case, qtimlvclassification does the following: a. Loads the submodule of the model. b. Produces results as video frames with classification labels. c. Sends them to the sinkpad of qtivcomposer.
qtivcomposer	1. Receives the original video stream with classification results on its sinkpads. 2. On its sourcepad, produces GST buffers with contents composed of video streams from its sinkpads.
Output	
Waylandsink	1. Receives the video in its sinkpad 2. Submits the video stream to Weston. 3. Weston renders the video stream and possible classifications generated for that scene on a local display device.

5.1.2.2 Image classification and encode with Neural Processing SDK

The use cases use the InceptionV3 image classification model with Qualcomm Neural Processing SDK to classify scenes from a single camera stream and either overlay or compose the classification labels. The streams are then encoded.

You can use any publicly available classification model with LiteRT and convert it to `.dlc` format. For instructions, see TensorFlow Model Conversion.

Use qtivoverlay plugin to apply classification overlay

Run the use case:

```
gst-launch-1.0 -e \
qtiqmmfsrc name=camsrc ! video/x-
raw,format=NV12_Q08C,width=1280,height=720,framerate=30/1 ! queue ! tee
name=split \
split. ! queue ! qtimetamux name=metamux ! queue ! qtivoverlay ! queue !
video/x-raw,format=NV12_Q08C,width=1280,height=720,interlace-
mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4 output-io-
mode=5 ! h264parse ! queue ! mp4mux ! queue ! filesink location=/etc/media/
video.mp4 \
split. ! queue ! qtimlvconverter ! queue ! qtimlsnpe delegate=dsp model=/etc/
models/inceptionv3.dlc ! queue ! qtimlvclassification threshold=40.0
results=2 module=mobilenet labels=/etc/labels/classification.labels ! text/x-
raw ! queue ! metamux.
```

To stop the use case, use **CTRL + C**.

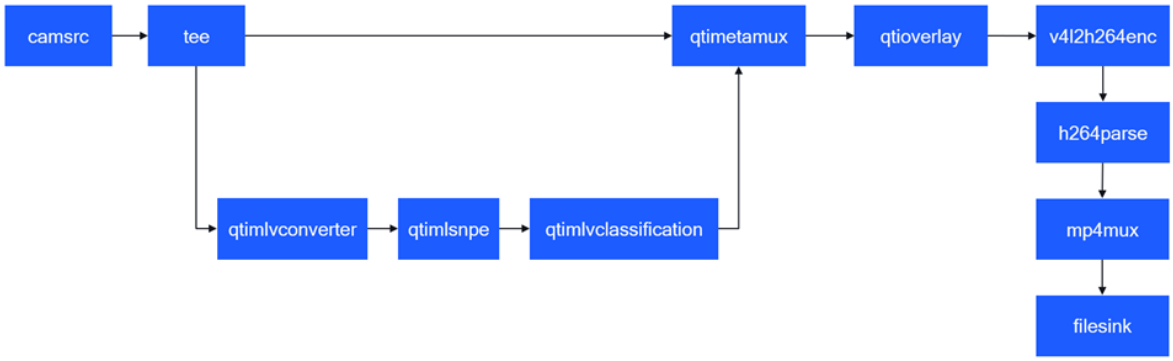


Figure 5-24 Pipeline for classification overlay and encode

The figure shows the flow of the use case execution:

1. Classify scenes from a video stream coming through a camera source.
2. Overlay the classification labels using overlaylib.
3. Encode this stream as an H.264 bitstream.
4. Multiplex the stream in an MP4 container and store it as an MP4 file.

The table provides the sequential processing stages of the pipeline execution:

Table 5-19 Pipeline processing stages for classification overlay and encode

Process	Description
Source	1. The video stream is collected from a camera source plugin and two copies are created: ▫ One stream is sent to the qtimetamux plugin to retain the video stream. ▫ The other stream is sent to an ML inferencing pipeline.
Preprocessing	
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs preprocessing: ▫ Color conversion ▫ Scaling down/up ▫ Normalization on the stream data when the model expects the floating point values as input 3. Converts the video stream to a tensor stream on its source pad. The classification model uses this tensor stream for inferencing.
Inferencing	
qtimlsnpe	1. Loads the model. 2. Modifies the graph for the chosen delegate. 3. Receives the tensor stream on its sinkpad. 4. Runs the inference and produces a tensor stream with the inference results on its source pad.
Postprocessing	
qtimlvclassification	1. Receives the inference tensors from a classification model on its sinkpad. 2. Converts the tensors into formats such as video or text that the multimedia plugins can process later. 3. Applies the threshold to the chosen number of results. 4. Loads the corresponding modules of the classification models. In this use case, qtimlvclassification does the following: a. Loads the submodule of the model. b. Produces results as video frames with classification labels. c. Sends them to the sinkpad of qtimetamux.
qtimetamux	1. Receives the video and text streams with the classification results corresponding to the video stream on its sinkpads. 2. Produces GST buffers with the contents of the video stream on its sink pad. 3. Adds classification result from the data sinkpad to GST buffer meta (meta muxing) on its source pad.
qtivoverlay	1. Receives the multiplexed stream. 2. Overlays the classification labels on the VideoFrame using CL. 3. Produces GST buffers with overlays in its source pad.
v4l2h264enc	1. Applies parameters to each frame of the video stream it's receiving on its sinkpad. 2. Encodes it into bitstream and sends it over its sourcepad.
h264parse	Adds more information about the bitstream to GStreamer buffer meta.
mp4mux	Receives these buffers and creates containers with format specification buffers.

Table 5-19 Pipeline processing stages for classification overlay and encode (cont.)

Process	Description
Output	
Filesink	Stores the resulting stream in a `/etc/media/video.mp4` file.
Playback	Pull `video.mp4` from the host computer and play it on a media player: `scp root@<IP address of target device>:/etc/ <destination directory>`

Use qtivcomposer to mix original frame with classification mask

Run the use case:

```
gst-launch-1.0 -e --gst-debug=2 \
qtiqmmfsrc name=camsrc ! video/x-
raw,format=NV12_Q08C,width=1280,height=720,framerate=30/1 ! queue ! tee
name=split \
split. ! queue ! qtivcomposer name=mixer sink_1::position="<30, 30>"
sink_1::dimensions="<320, 180>" ! queue ! video/x-
raw,format=NV12,width=1920,height=1080,interlace-
mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4 output-io-
mode=5 ! h264parse ! queue ! mp4mux ! queue ! filesink location=/etc/media/
video.mp4 \
split. ! queue ! qtimlvconverter ! queue ! qtimlsnpe delegate=dsp model=/etc/
models/inceptionv3.dlc ! queue ! qtimlvclassification threshold=40.0
results=2 module=mobilenet labels=/etc/labels/classification.labels ! video/x-
raw,format=BGRA,width=640,height=360 ! queue ! mixer.
```

To stop the use case, use **CTRL + C**.

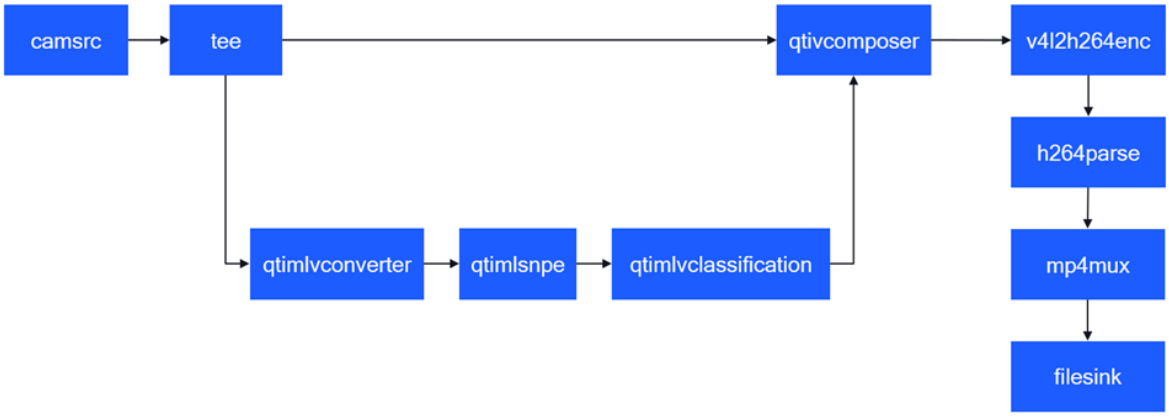


Figure 5-25 Pipeline for classification and encode with qtivcomposer

The figure shows the flow of the use case execution:

- Classify scenes from a video stream coming through a camera source.
- Compose classification labels and video stream together using qtivcomposer.

- Encode this stream as an H.264 bitstream.
- Multiplex the stream in an MP4 container and store it as an MP4 file.

The table provides the sequential processing stages of the pipeline execution:

Table 5-20 Pipeline processing stages for classification and encode with qtivcomposer

Process	Description
Source	1. The video stream is collected from a camera source plugin and two copies are created: ▫ One stream is sent to the qtivcomposer plugin to retain the video stream. ▫ The other stream is sent to an ML inferencing pipeline.
Preprocessing	
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs preprocessing: ▫ Color conversion ▫ Scaling down/up ▫ Normalization on the stream data when the model expects the floating point values as input 3. Converts the video stream to a tensor stream on its source pad. The classification model uses this tensor stream for inferencing.
Inferencing	
qtimlsnpe	1. Loads the model. 2. Modifies the graph for the chosen delegate. 3. Receives the tensor stream on its sinkpad. 4. Runs the inference and produces a tensor stream with the inference results on its source pad.
Postprocessing	
qtimlvclassification	1. Receives the inference tensors from a classification model on its sinkpad. 2. Converts the tensors into formats such as video or text that the multimedia plugins can process later. 3. Applies the threshold to the chosen number of results. 4. Loads the corresponding modules of the classification models. In this use case, qtimlvclassification does the following: a. Loads the submodule of the model. b. Produces results as video frames with classification labels. c. Sends them to the sinkpad of qtivcomposer.
qtivcomposer	1. Receives original video stream and video stream with classification results on its sinkpads. 2. On its sourcepad, produces GST buffers with the contents composed of video streams from its sinkpads.
v4l2h264enc	1. Applies parameters to each frame of the video stream it's receiving on its sinkpad. 2. Encodes it into bitstream and sends it over its sourcepad.
h264parse	Adds more information about the bitstream to GStreamer buffer meta.
mp4mux	Receives these buffers and creates containers with format specification buffers.
Output	

Table 5-20 Pipeline processing stages for classification and encode with qtivcomposer (cont.)

Process	Description
Filesink	Stores the resulting stream in a `/etc/media/video.mp4` file.
Playback	Pull `video.mp4` from the host computer and play it on a media player: `scp root@<IP address of target device>:/etc/ <destination directory>`

5.1.2.3 Object detection and display with Neural Processing SDK

The use cases use a `yolonas.dlc` object detection model with Qualcomm Neural Processing SDK to identify an object from a camera stream. The use case is to overlay or compose the bounding boxes over the detected objects, and the display the results.

Use qtivoverlay plugin to apply bounding box overlay

Run the use case:

```
gst-launch-1.0 -e \
qtiqmmfsrc name=camsrc ! video/x-
raw,format=NV12_Q08C,width=1280,height=720,framerate=30/1 ! queue ! tee
name=split \
split. ! queue ! qtimetamux name=metamux ! queue ! qtivoverlay ! queue !
waylandsink fullscreen=true sync=false \
split. ! queue ! qtimlvconverter ! queue ! qtimlsnpe delegate=dsp model=/etc/
models/yolonas.dlc layers="</heads/Mul, /heads/Sigmoid>" ! queue ! \
qtimlvdetection threshold=51.0 results=10 module=yolo-nas labels=/etc/labels/
yolonas.labels ! text/x-raw ! queue ! metamux.
```

To stop the use case, use **CTRL + C**.

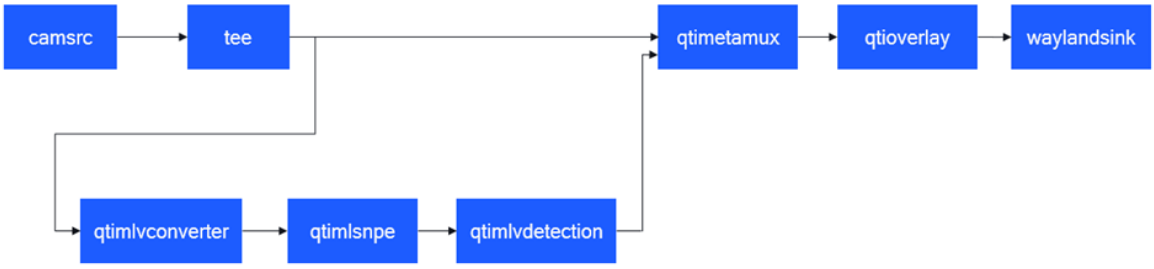


Figure 5-26 Pipeline for bounding box overlay

The figure shows the flow of the use case execution:

- Identify an object in a scene from a video stream coming through camera source.
- Overlay the bounding boxes over the detected objects using overlaylib.
- Display the results on a local display.

The table provides the sequential processing stages of the pipeline execution:

Table 5-21 Pipeline processing stages for bounding box overlay

Process	Description
qtiqmmfsrc	1. Collects the video stream (source) and creates two copies of the source: ▫ One stream is sent to qtimetamux plugin to retain the video stream. ▫ The other stream is sent to an ML inferencing pipeline.
Preprocessing	
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs preprocessing: ▫ Color conversion ▫ Scaling down/up ▫ Normalization on the stream data when the model expects the floating point values as input 3. Converts the video stream to a tensor stream on its source pad. The object detection model uses this tensor stream for inferencing.
Inferencing	
qtimlsnpe	1. Loads the object detection model. 2. Modifies the graph for the chosen delegate. 3. Receives the tensor stream on its sinkpad. 4. Runs the inference and produces a tensor stream with the object detection results on its source pad.
Postprocessing	
qtimlvdetection	1. Receives the inference tensors from the object detection model. 2. Converts the inference tensors on its sinkpad into formats like video or text that the multimedia plugins can process later. 3. Applies the threshold to the chosen number of results. 4. Loads the corresponding modules for detection models. In this use case, qtimlvdetection does the following: a. Loads YOLO-NAS submodule. b. Produces results as structures of text. c. Sends them to the sinkpad of qtimetamux.
qtimetamux	1. Receives video stream and text stream with bounding box results corresponding to the video stream on its sinkpads. 2. Produces GST buffers with contents of the video stream from its sink pad. 3. Adds the bounding boxes as `GstVideoRegionOfInterest` from data sinkpad to GST buffers meta (meta muxing) on its source pad.
qtivoverlay	1. Receives the multiplexed stream. 2. Overlays the bounding boxes on the VideoFrame using CL. 3. Produces GST buffers with overlays in its source pad.
Output	
Waylandsink	1. Receives the video stream on its sinkpad. 2. Submits the video stream to Weston. 3. Weston renders the video stream and bounding boxes generated for the objects in that scene on a local display device.

Use qtivcomposer to mix original frame with bounding box mask

Run the use case:

```
gst-launch-1.0 -e --gst-debug=2 \
qtiqmmfsrc name=camsrc ! video/x-
raw,format=NV12_Q08C,width=1280,height=720,framerate=30/1 ! queue ! tee
name=split \
split. ! queue ! qtivcomposer name=mixer sink_1::dimensions="<1920,1080>" !
queue ! waylandsink fullscreen=true sync=false \
split. ! queue ! qtimlvconverter ! queue ! qtimlsnpe delegate=dsp model=/etc/
models/yolonas.dlc layers="</heads/Mul, /heads/Sigmoid>" ! queue ! \
qtimlvdetection threshold=51.0 results=10 module=yolo-nas labels=/etc/labels/
yolonas.labels ! video/x-raw,width=640,height=360 ! queue ! mixer.
```

To stop the use case, use **CTRL + C**.

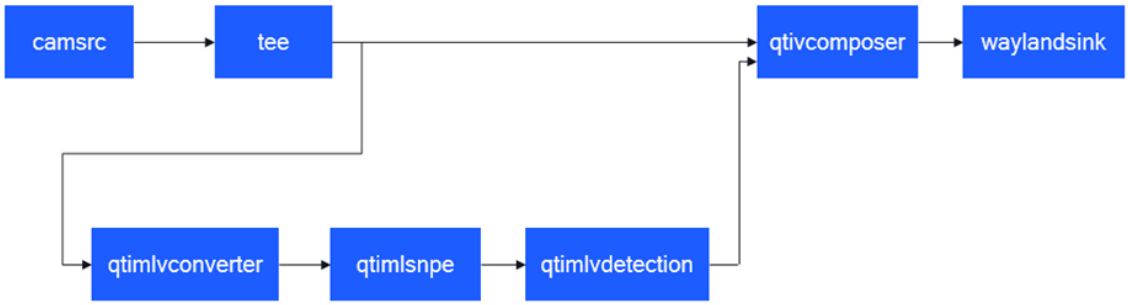


Figure 5-27 Pipeline for bounding box mask with qtivcomposer

The figure shows the flow of the use case execution:

1. Identifies object scenes in the scene from a video stream, which is coming through a camera source.
2. Composes the following using qtivcomposer:
   a. Bounding boxes over objects detected.
   b. Original video stream.
3. Display the results.

The table provides the sequential processing stages of the pipeline execution:

Table 5-22 Pipeline processing stages for bounding box mask with qtivcomposer

Process	Description
qtiqmmfsrc	1. Collects the video stream (source) and creates two copies of the source: a. One stream is sent to qtivcomposer plugin to retain the video stream. b. The other stream is sent to an ML inferencing pipeline.
Preprocessing	

Table 5-22 Pipeline processing stages for bounding box mask with qtivcomposer (cont.)

Process	Description
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs preprocessing: ▫ Color conversion. ▫ Scaling down/up. ▫ Normalization on the stream data when the model expects the floating point values as input. 3. Converts the video stream to a tensor stream on its source pad. The object detection model uses this tensor stream for inferencing.
Inferencing	
qtimlsnpe	1. Loads the object detection model. 2. Modifies the graph for the chosen delegate. 3. Receives the tensor stream on its sinkpad. 4. Runs the inference and produces a tensor stream with the object detection results on its source pad.
Postprocessing	
qtimlvdetection	1. Receives the inference tensors from the objection detection model. 2. Converts the inference tensors on its sinkpad into formats like video or text that the multimedia plugins can process later. 3. Applies the threshold to the chosen number of results. 4. Loads the corresponding modules for detection models. In this use case, qtimlvdetection does the following: a. Loads the YOLO-NAS submodule. b. Produces video frames with only bounding boxes that can be overlaid on objects. c. Sends them to the sinkpad of qtivcomposer.
qtivcomposer	1. Receives the original video stream and video stream with bounding boxes on its sinkpads. 2. On its sourcepads, produces content that's composed of the video streams processed from its sinkpads.
Output	
Waylandsink	1. Receives the video stream on its sinkpad. 2. Submits the video stream to Weston. 3. Weston displays the following on a local display device: a. The video stream is captured from the camera. b. The bounding boxes are drawn over the allowed number of objects identified in that scene.

5.1.2.4 Object detection and encode with Neural Processing SDK

The use cases use a `yolonas.dlc` object detection model with Qualcomm Neural Processing SDK to identify an object from a camera stream. The use case is to overlay or compose the bounding boxes over the detected objects, and then encode the stream as a H.264 bitstream.

Use qtivoverlay plugin to apply detection overlay

Run the use case:

```
gst-launch-1.0 -e \
qtiqmmfsrc name=camsrc ! video/x-
raw,format=NV12_Q08C,width=1280,height=720,framerate=30/1 ! queue ! tee
name=split \
split. ! queue ! qtimetamux name=metamux ! queue ! qtivoverlay ! queue !
v4l2h264enc capture-io-mode=4 output-io-mode=5 ! h264parse ! queue ! mp4mux !
queue ! filesink location=/etc/media/video.mp4 \
split. ! queue ! qtimlvconverter ! queue ! qtimlsnpe delegate=dsp model=/etc/
models/yolonas.dlc layers="</heads/Mul, /heads/Sigmoid>" ! queue ! \
qtimlvdetection threshold=51.0 results=10 module=yolo-nas labels=/etc/labels/
yolonas.labels ! text/x-raw ! queue ! metamux.
```

To stop the use case, use **CTRL + C**.

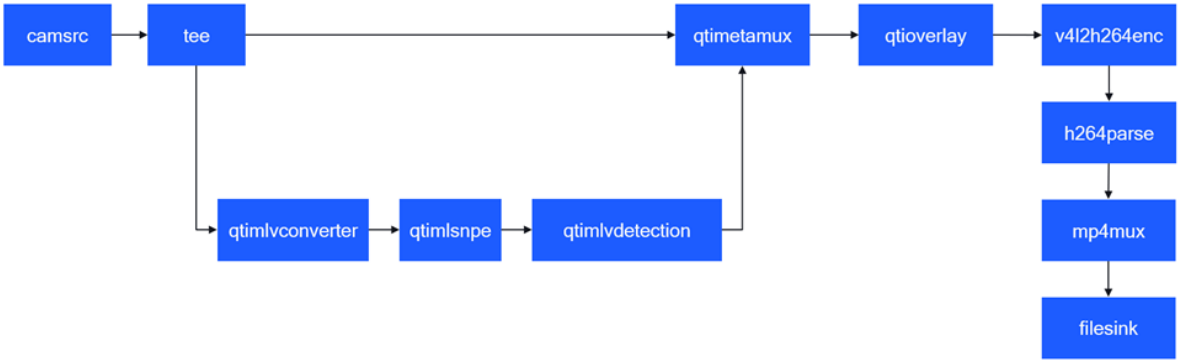


Figure 5-28 Pipeline for bounding box overlay and encode

The figure shows the flow of the use case execution:

1. Identify object scenes from a video stream, which is coming through a camera source.
2. Overlay bounding boxes over the detected objects using overlaylib.
3. Encode this stream as an H.264 bitstream.
4. Multiplex the stream in an MP4 container and stored as an MP4 file.

The table provides the sequential processing stages of the pipeline execution:

Table 5-23 Pipeline processing stages for bounding box overlay and encode

Process	Description
qtiqmmfsrc	1. Collects the video stream (source) and creates two copies of the source: ▫ One stream is sent to qtimetamux plugin to retain the video stream. ▫ The other stream is sent to an ML inferencing pipeline.
Preprocessing	
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs preprocessing: ▫ Color conversion ▫ Scaling down/up ▫ Normalization on the stream data when the model expects the floating point values as input 3. Converts the video stream to a tensor stream on its source pad. The object detection model uses this tensor stream for inferencing.
Inferencing	
qtimlsnpe	1. Loads the object detection model. 2. Modifies the graph for the chosen delegate. 3. Receives the tensor stream on its sinkpad. 4. Runs the inference and produces a tensor stream with the object detection results on its source pad.
Postprocessing	
qtimlvdetection	1. Receives the inference tensors from the object detection model. 2. Converts the inference tensors on its sinkpad into formats like video or text that the multimedia plugins can process later. 3. Applies the threshold to the chosen number of results. 4. Loads the corresponding modules for detection models. In this use case, qtimlvdetection does the following: a. Loads the YOLO-NAS submodule. b. Produces results as structures of text. c. Sends them to the sinkpad of qtimetamux.
qtimetamux	1. Receives video stream and text stream with the bounding box results corresponding to the video stream on its sinkpads. 2. Produces GST buffers with the contents of the video stream from its sink pad. 3. Adds the bounding boxes as `GstVideoRegionOfInterest` from data sinkpad to GST buffers meta (meta muxing) on its source pad.
qtivoverlay	1. Receives the multiplexed stream. 2. Overlays the bounding boxes on the VideoFrame using CL. 3. Produces GST buffers with overlays in its source pad.
v4l2h264enc	1. Applies parameters to each frame of the video stream it's receiving on its sinkpad. 2. Encodes it into bitstream and sends it over its sourcepad.
h264parse	Adds more information about the bitstream to the GStreamer buffer meta.
mp4mux	Receives these buffers and creates containers with format specification buffers.

Table 5-23 Pipeline processing stages for bounding box overlay and encode (cont.)

Process	Description
Output	
Filesink	Stores the resulting stream in a `/etc/media/video.mp4` file.
Playback	Pull `video.mp4` from the host computer and play it on a media player: `scp root@<IP address of target device>:/etc/ <destination directory>`

Use qtivcomposer to mix original frame with detection mask

Run the use case:

```
gst-launch-1.0 -e \
qtiqmmfsrc name=camsrc ! video/x-
raw,format=NV12_Q08C,width=1280,height=720,framerate=30/1 ! queue ! tee
name=split \
split. ! queue ! qtivcomposer name=mixer ! queue ! video/x-
raw,format=NV12,width=1920,height=1080,interlace-
mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4 output-io-
mode=5 ! h264parse ! queue ! mp4mux ! queue ! filesink location=/etc/media/
video.mp4 \
split. ! queue ! qtimlvconverter ! queue ! qtimlsnpe delegate=dsp model=/etc/
models/yolonas.dlc layers="</heads/Mul, /heads/Sigmoid>" ! queue !
qtimlvdetection threshold=51.0 results=10 module=yolo-nas labels=/etc/labels/
yolonas.labels ! video/x-raw,width=640,height=360 ! queue ! mixer.
```

To stop the use case, use **CTRL + C**.

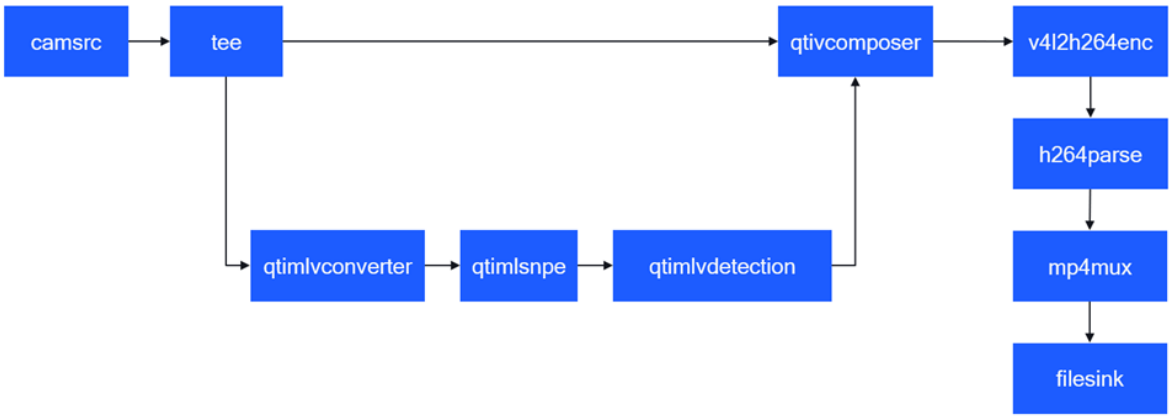


Figure 5-29 Pipeline for bounding box mask and encode with qtivcomposer

The figure shows the flow of the use case execution:

1. Identify object scenes from a video stream, which is coming through a camera source.
2. Compose bounding boxes over objects detected and original video stream using qtivcomposer.

3. Encode this stream as an H.264 bitstream.
4. Multiplex the stream in an MP4 container and stored as an MP4 file.

The table provides the sequential processing stages of the pipeline execution:

Table 5-24 Pipeline processing stages for bounding box mask and encode with qtivcomposer

Process	Description
qtiqmmfsrc	1. Collects the video stream (source) and creates two copies of the source: ▫ One stream is sent to qtimetamux plugin to retain the video stream. ▫ The other stream is sent to an ML inferencing pipeline.
Preprocessing	
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs preprocessing: ▫ Color conversion ▫ Scaling down/up ▫ Normalization on the stream data when the model expects the floating point values as input 3. Converts the video stream to a tensor stream on its source pad. The object detection model uses this tensor stream for inferencing.
Inferencing	
qtimlsnpe	1. Loads the object detection model. 2. Modifies the graph for the chosen delegate. 3. Receives the tensor stream on its sinkpad. 4. Runs the inference and produces tensor stream with the object detection results on its source pad.
Postprocessing	
qtimlvdetection	1. Receives the inference tensors from the object detection model. 2. Converts the inference tensors on its sinkpad into formats like video or text that the multimedia plugins can process later. 3. Applies the threshold to the chosen number of results. 4. Loads the corresponding modules for detection models. In this use case, qtimlvdetection does the following: a. Loads the YOLO-NAS submodule. b. Produces video frames with only bounding boxes that can be overlaid on objects. c. Sends them to sinkpad of qtivcomposer.
qtivcomposer	1. Receives the original video stream and video stream with bounding boxes on its sinkpads 2. On its sourcepads, produces content that's composed of the video streams processed from its sinkpads.
v4l2h264enc	1. Applies parameters to each frame of the video stream its receiving on its sinkpad. 2. Encodes it into bitstream and sends it over its sourcepad.
h264parse	Adds more information about the bitstream to the GStreamer buffer meta.
mp4mux	Receives these buffers and creates containers with format specification buffers.

Table 5-24 Pipeline processing stages for bounding box mask and encode with qtivcomposer (cont.)

Process	Description
Output	
Filesink	Stores the resulting stream in a `/etc/media/video.mp4` file.
Playback	Pull `video.mp4` from the host computer and play it on a media player: `scp root@<IP address of target device>:/etc/ <destination directory>`

5.1.2.5 Image segmentation and display with Neural Processing SDK

The use case uses the DeepLab v3 model with the Qualcomm Neural Processing SDK runtime to identify the semantic segmentations in a scene from a camera stream. The use case is to compose the semantics and the video stream together using qtivcomposer, and then display the results.

Use qtivcomposer to mix original frame with segmentation mask

Run the use case:

```
gst-launch-1.0 -e --gst-debug=2 \
qtiqmmfsrc name=camsrc ! video/x-
raw,format=NV12_Q08C,width=1920,height=1080,framerate=30/1 ! queue ! tee
name=split  \
split. ! queue ! qtivcomposer name=mixer sink_1::dimensions="<1920,1080>"
sink_1::alpha=0.5 ! queue ! waylandsink fullscreen=true \
split. ! queue ! qtimlvconverter ! queue ! qtimlsnpe delegate=dsp model=/etc/
models/deeplabv3_resnet50.dlc ! queue ! qtimlvsegmentation module=deeplab-
argmax labels=/etc/labels/deeplabv3_resnet50.labels ! video/x-
raw,width=640,height=360 ! queue ! mixer.
```

To stop the use case, use **CTRL + C**.

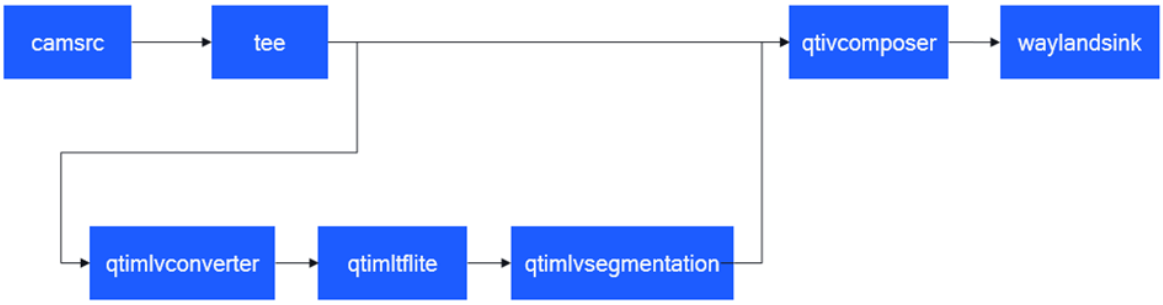


Figure 5-30 Pipeline for segmentation with qtivcomposer

The figure shows the flow of the use case execution:

1. Identify scenes from a video stream coming through a camera source.
2. Compose semantic segmentation and video stream using qtivcomposer.
3. Display the results.

The table provides the sequential processing stages of the pipeline execution:

Table 5-25 Pipeline processing stages for segmentation with qtivcomposer

Process	Description
qtiqmmfsrc	1. Collects the video stream (source) and creates two copies of the source: ▫ One stream is sent to the qtivcomposer plugin to retain the video stream. ▫ The other stream is sent to the ML inferencing branch in the pipeline.
Preprocessing	
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs preprocessing: ▫ Color conversion ▫ Scaling down/up ▫ Normalization on the stream data when the model expects the floating point values as input 3. Converts the video stream to a tensor stream on its source pad. The segmentation model uses this tensor stream for inferencing.
Inferencing	
qtimlsnpe	1. Loads the segmentation model. 2. Modifies the graph for the chosen delegate. 3. Receives the tensor stream on its sinkpad. 4. Runs the inference and produces a tensor stream with the segmentation results on its source pad.
Postprocessing	
qtimlvsegmentation	1. Receives the inference tensors on its sinkpad. 2. Converts the inference tensors into video formats that the multimedia plugins can process later. 3. Produces the semantic segmentations for the frame. 4. Loads the corresponding modules for the segmentation models. In this use case, qtimlvsegmentation does the following: a. Loads the deeplab-argmax submodule. b. Produces video frames with segmentation masks. c. Sends them to the sinkpad of qtivcomposer.
qtivcomposer	1. Receives the original video stream with segmentation mask on its sinkpads. 2. On its sourcepad, produces GST buffers with contents composed of video streams from its sinkpads.

Table 5-25 Pipeline processing stages for segmentation with qtivcomposer (cont.)

Process	Description
Output	
Waylandsink	1. Receives the video stream on its sinkpad. 2. Submits the video stream to Weston. 3. Weston displays the following on the local display device: ▫ The video stream that's captured from the camera. ▫ The segmentation masks that are drawn over objects/components in that scene.

5.1.2.6 Image segmentation and encode with Neural Processing SDK

The use case uses the DeepLab v3 model with the Qualcomm Neural Processing SDK runtime. The use case is to compose the semantic segmentations and original video stream, encode this stream, and then multiplex it in an MP4 container.

Run the use case:

```
gst-launch-1.0 -e --gst-debug=2 \
qtiqmmfsrc name=camsrc ! video/x-
raw,format=NV12_Q08C,width=1920,height=1080,framerate=30/1 ! queue ! tee
name=split \
split. ! queue ! qtivcomposer name=mixer sink_1::dimensions="<1920,1080>"
sink_1::alpha=0.5 ! queue ! video/x-
raw,format=NV12,width=1920,height=1080,interlace-
mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4 output-io-
mode=5 ! h264parse ! queue ! mp4mux ! queue ! filesink location=/etc/media/
video.mp4 \
split. ! queue ! qtimlvconverter ! queue ! qtimlsnpe delegate=dsp model=/etc/
models/deeplabv3_resnet50.dlc ! queue ! qtimlvsegmentation module=deeplab-
argmax labels=/etc/labels/deeplabv3_resnet50.labels ! video/x-
raw,width=640,height=360 ! queue ! mixer.
```

To stop the use case, use **CTRL + C**.

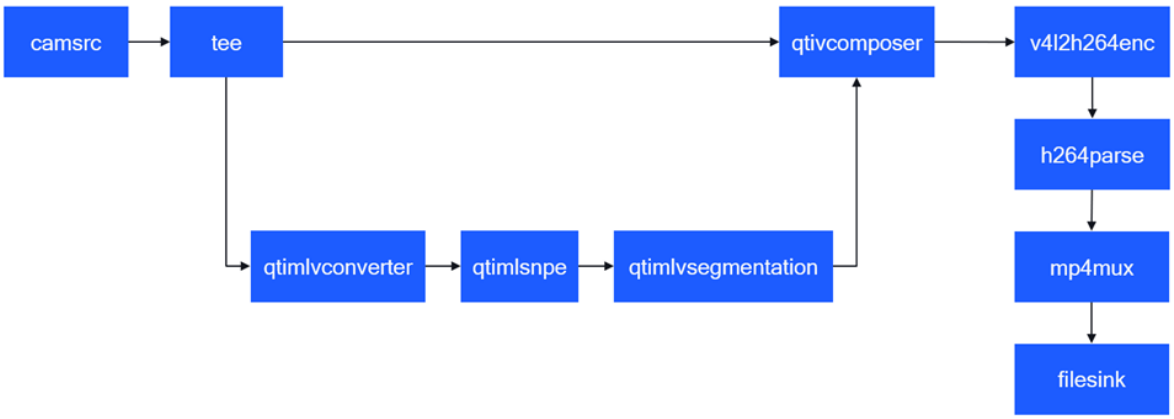


Figure 5-31 Pipeline for image segmentation and encode with qtivcomposer

The figure shows the flow of the use case execution:

1. Identify scenes from a video stream coming through a camera source.
2. Compose semantic segmentation and video stream using qtivcomposer.
3. Encode the stream as an H.264 bit stream and multiplex the stream in an MP4 container.

The table provides the sequential processing stages of the pipeline execution:

Table 5-26 Pipeline processing stages for image segmentation and encode with qtivcomposer

Process	Description
qtiqmmfsrc	1. Collects the video stream (source) and creates two copies of the source: ▫ One stream is sent to the qtivcomposer plugin to retain the video stream. ▫ The other stream is sent to an ML inferencing branch in the pipeline.
Preprocessing	
qtimlvconverter	1. Receives the video stream on its sink pad. 2. Performs preprocessing: ▫ Color conversion ▫ Scaling down/up ▫ Normalization on the stream data when the model expects the floating point values as input 3. Converts the video stream to a tensor stream on its source pad. The segmentation model uses this tensor stream for inferencing.
Inferencing	
qtimlsnpe	1. Loads the segmentation model. 2. Modifies the graph for the chosen delegate. 3. Receives the tensor stream on its sinkpad. 4. Runs the inference and produces a tensor stream with the segmentation results on its source pad.
Postprocessing	
qtimlvsegmentation	1. Receives the inference tensors on its sinkpad. 2. Converts the inference tensors into video formats that the multimedia plugins can process later. 3. Produces the semantic segmentations for the frame. 4. Loads the corresponding modules for the segmentation models. In this use case, qtimlvsegmentation does the following: a. Loads deeplab-argmax submodule. b. Produces video frames with segmentation masks. c. Sends them to the sinkpad of qtivcomposer.
qtivcomposer	1. Receives the original video stream with segmentation mask on its sinkpads. 2. Produces on its sourcepad GST buffers with contents composed of video streams from its sinkpads.
v4l2h264enc	1. Applies parameters to each frame of the video stream it's receiving on its sinkpad. 2. Encodes it into bitstream and sends it over its sourcepad.
h264parse	Adds more information about the bitstream to the GStreamer buffer meta.

Table 5-26 Pipeline processing stages for image segmentation and encode with qtivcomposer (cont.)

Process	Description
mp4mux	Receives these buffers and creates containers format specification buffers.
Output	
Filesink	Stores the resulting stream in a `/etc/media/video.mp4` file.
Playback	Pull `video.mp4` from the host computer and play it on a media player: `scp root@<IP address of target device>:/etc/ <destination directory>`

5.2 Run multimedia use cases

The multimedia use cases show various multimedia scenarios using the GStreamer pipelines.

Prerequisites

After completing the prerequisites in GStreamer command-line use cases, ensure that for video/audio playback, you push a pre-recorded video on the device or record a test video/audio.

Push the files on the target:

```
scp -r <file_name> root@[DEVICE IP-ADDR]:/opt
```

NOTE GBM errors are non-fatal. However, to avoid these errors, use the following commands:

```
mkdir -p /var/cache/display
echo 0 > /var/cache/display/gbm_dbg_cfg.txt
```

NOTE For **QCS9075** and **QCS8275**, set the `fullscreen` property to *`true`* for the Waylandsink plugin: `fullscreen=true`.

See also

- Camera Guide
- Video Guide
- Audio Guide

5.2.1 Camera

The Qualcomm IM SDK camera plugin, camsrc, acts as a client to the Qualcomm camera service. The camera use cases show the execution and display of the 1080p YUV streams from a live source.

For more information about the camera plugin, see Camera architecture.

Prerequisites

- Complete the preconditions mentioned in Run multimedia use cases.
- Configure the display:

```
mount -o remount,rw /
```

```
export XDG_RUNTIME_DIR=/dev/socket/weston && export
WAYLAND_DISPLAY=wayland-1
```

Single 1080p YUV stream from live source

Run the use case:

```
gst-launch-1.0 -e qtiqmmfsrc name=camsrc video_1::type=preview ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! waylandsink fullscreen=true sync=false
```

The content from the camera stream is displayed.

To stop the use case, use **CTRL + C**.

The pipeline shows a single 1080p stream taken from the camera and sent to display.

Three 720 YUV streams from live source

Run the use case:

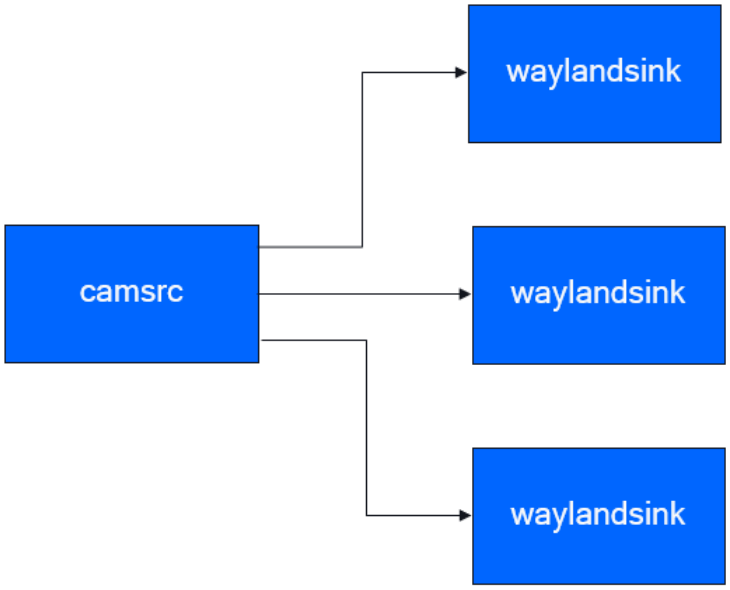
```
gst-launch-1.0 -e qtivcomposer name=mixer sink_0::position="<0, 0>"
sink_0::dimensions="<480, 270>" sink_1::position="<480, 0>"
sink_1::dimensions="<480, 270>" sink_2::position="<960, 0>"
sink_2::dimensions="<480, 270>" mixer. ! queue ! waylandsink fullscreen=true
qtiqmmfsrc name=camsrc camera=0 video_0::type=preview video_1::type=video !
video/x-raw,format=NV12_Q08C,width=1280,height=720,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! queue ! mixer. camsrc. ! video/x-
raw,format=NV12,width=1280,height=720,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! queue ! mixer. camsrc. ! video/x-
raw,format=NV12,width=1280,height=720,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! queue ! mixer.
```

The content from camera streams is displayed.

To stop the use case, use **CTRL + C**.

NOTE With the GMSL camera, it's been observed that the display doesn't work and the use case doesn't stop when using **CTRL + C**.

The pipeline shows three 720p streams taken from the camera and sent to display with each stream displayed at different positions on the screen.



Request many streams with different fps from camera

- Two streams with capture request mode at 30 fps/15 fps:

```
gst-launch-1.0 -e --gst-debug=fpsdisplaysink:6 qtiqmmfsrc name=qmmf frc-
mode=capture-request video_0::type=video video_1::type=preview ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1 ! fpsdisplaysink
name=disp1 signal-fps-measurements=true text-overlay=false video-
sink="fakesink" qmmf. ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=15/1 ! fpsdisplaysink
name=disp2 signal-fps-measurements=true text-overlay=false video-
sink="fakesink"
```

- Two streams with capture request mode at 17 fps/10 fps:

```
gst-launch-1.0 -e --gst-debug=fpsdisplaysink:6 qtiqmmfsrc name=qmmf frc-
mode=capture-request video_0::type=video video_1::type=preview
qmmf.video_0 ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=17/1 ! fpsdisplaysink
name=disp1 signal-fps-measurements=true text-overlay=false video-
sink="fakesink" qmmf.video_1 ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=10/1 ! fpsdisplaysink
name=disp2 signal-fps-measurements=true text-overlay=false video-
sink="fakesink"
```

- Two streams with capture request mode at 5 fps/25 fps:

  The smaller frame rate is processed first and the camera updates the frame rate on the second stream.

  ```
  gst-launch-1.0 -e --gst-debug=fpsdisplaysink:6 qtiqmmfsrc name=qmmf frc-mode=capture-request video_0::type=video video_1::type=preview qmmf.video_0 ! video/x-raw,format=NV12,width=1920,height=1080,framerate=5/1 ! fpsdisplaysink name=disp1 signal-fps-measurements=true text-overlay=false video-sink="fakesink" qmmf.video_1 ! video/x-raw,format=NV12,width=1920,height=1080,framerate=25/1 ! fpsdisplaysink name=disp2 signal-fps-measurements=true text-overlay=false video-sink="fakesink"
  ```

Set sensor mode

Set the sensor mode in the camsrc plugin:

```
export GST_PLUGIN_PATH=/usr/lib/gstreamer-1.0 && gst-launch-1.0 qtiqmmfsrc camera=0 name=camsrc sensor-mode=0 ! video/x-raw,format=NV12,width=1920,height=1080,framerate=30/1 ! multifilesink enable-last-sample=false location="/opt/frame%d.yuv" max-files=5
```

For `sensor-mode=<option>`, add the appropriate number for the sensor mode.

For example, for `imx766_preview_1080p30fps_streaming_sensor_mode_1` use case, run the following command to set the sensor mode as 0:

```
export GST_PLUGIN_PATH=/usr/lib/gstreamer-1.0 && gst-launch-1.0 qtiqmmfsrc camera=0 name=camsrc sensor-mode=0 ! video/x-raw,format=NV12,width=1920,height=1080,framerate=30/1 ! multifilesink enable-last-sample=false location="/opt/frame%d.yuv" max-files=5
```

5.2.1.1 Single camera stream with socket

This use case shows how to store a camera stream in a socket. This camera stream from the socket is then encoded to AVC and stored in a file.

1. Complete the preconditions mentioned in Run multimedia use cases.
2. Run the use case:
   - In console 1:

     ```
     gst-launch-1.0 -e qtisocketsrc socket=/tmp/input.sock ! video/x-raw,format=NV12,width=1920,height=1080,framerate=30/1,interlace-mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4 output-io-mode=5 ! h264parse ! mp4mux ! queue ! filesink location=/opt/video.mp4
     ```

   - In console 2:

     ```
     gst-launch-1.0 -e qtiqmmfsrc name=camsrc ! video/x-raw,format=NV12,width=1920,height=1080,framerate=30/1 ! qtisocketsink socket=/tmp/input.sock
     ```

The figure shows the flow of the pipeline execution:

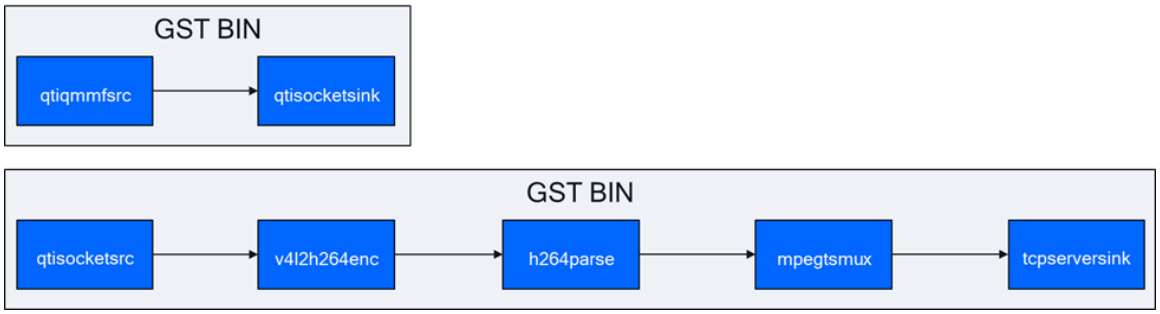


Figure 5-32 1080p camera stream sent to socket, read from socket, encoded to AVC and stored in file

5.2.1.2 Troubleshooting

For any application, if the camera service doesn't work or fails during execution, see Camera troubleshooting to restart it.

5.2.2 Video encode and decode

The camera and video encode use cases show various scenarios for encoding and processing the live 4K and 1080p camera streams.

Before you begin running the use cases, ensure that you complete the preconditions mentioned in Run multimedia use cases.

NOTE 24 end-to-end concurrent video sessions aren't supported for QCS6490.

5.2.2.1 One stream–1080p AVC RTSP from live source

The use case uses a 1080p stream from the camera for encoding. The encoded stream is sent over the network using the RTSP streaming.

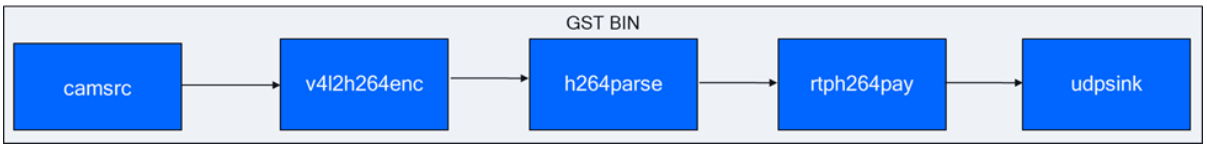


Figure 5-33 Pipeline for encoding 1080p RTSP stream

- Run the RTSP server on target with udpsrc (You can run it in the background as a service):

```
gst-rtsp-server -p 8900 -a "< IP_address of the device >"  -m /live
"( udpsrc name=pay0 port=8554 caps=\"application/x-rtp,media=video,clock-
rate=90000,encoding-name=H264,payload=96\" )"
```

- Run the pipeline in another console on the target:

```
gst-launch-1.0 -e qtiqmmfsrc name=camsrc ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! queue ! v4l2h264enc capture-io-mode=4
```

```
output-io-mode=5 ! queue ! h264parse config-interval=-1 ! rtph264pay
pt=96 ! udpsink host=< IP_address of the device > port=8554
```

The streamed video is displayed as an RTSP stream on a host.

To stop the use case, use **CTRL + C**.

View RTSP stream on host computer

Prerequisites:

- Install the VLC media player on the host computer.
- Set the binary/executable paths in the environment variables.

On the Linux host, do one of the following:

- `vlc -vvv rtsp://<IP_address of the device>:8900/live` [Ubuntu 18.04 with VLC version 3.0.8]
- `ffplay -rtsp_transport tcp rtsp://<IP_address of the device>:8900/live`

On the Windows host, do the following:

1. Open the VLC media player.
2. Go to **Media** > **Open Network Stream**, or use **CTRL + N**.
3. Enter `rtsp://<IP_address of the device>:8900/live`.
4. Select **Play**.

5.2.2.2 Two streams–4K AVC and 480p AVC from live source

The use cases use one 4k stream and one 1080p stream from the camera for encoding. Each encoded stream is multiplexed into a different file.

Run the use case:

```
gst-launch-1.0 -e qtiqmmfsrc name=camsrc video_0::type=preview ! video/x-
raw,format=NV12_Q08C,width=3840,height=2160,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! queue ! v4l2h264enc capture-io-mode=4
output-io-mode=5 ! queue ! h264parse ! mp4mux ! queue ! filesink
location="/opt/mux1.mp4" camsrc. ! video/x-
raw,format=NV12_Q08C,width=640,height=480,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! queue ! v4l2h264enc capture-io-mode=4
output-io-mode=5 ! queue ! h264parse ! mp4mux ! queue ! filesink
location="/opt/mux2.mp4"
```

To stop the use case, use **CTRL + C**. Pull the recorded content from the device using the following `scp` command and then play the content on a host PC.

You can play the MP4 file on a media player. Pull the recorded content:

```
scp -r root@[DEVICE IP-ADDR]:/opt/mux1.mp4 user1@[HOST IP-ADDR]:/workspace/
```

```
scp -r root@[DEVICE IP-ADDR]:/opt/mux2.mp4 user1@[HOST IP-ADDR]:/workspace/
```

The figure shows the flow of the pipeline execution:

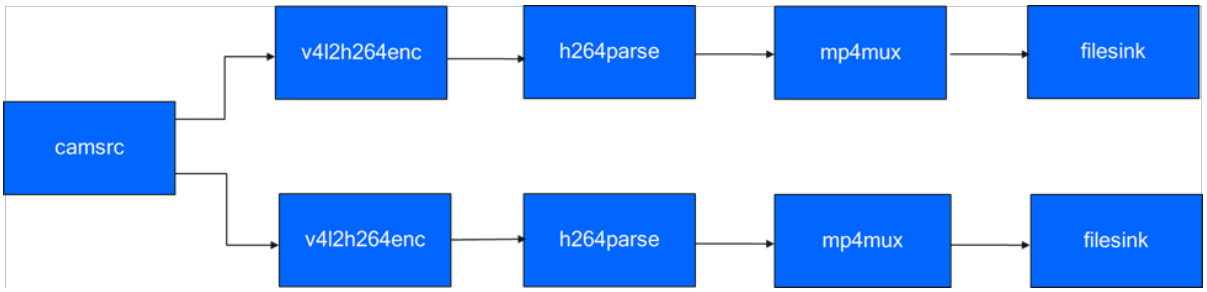


Figure 5-34 Pipeline for encoding two (4k and 1080p) camera streams

5.2.2.3 Three 1080p AVC streams from live source

The use case uses three 1080p streams from the camera for encoding. Each encoded stream is multiplexed into a different file.

Run the pipeline:

```
gst-launch-1.0 -e qtiqmmfsrc name=camsrc video_0::type=preview ! video/x-
raw,format=NV12_Q08C,width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! queue ! v4l2h264enc capture-io-mode=4
output-io-mode=5 ! queue ! h264parse ! mp4mux ! queue ! filesink
location="/opt/mux1.mp4" camsrc. ! video/x-
raw,format=NV12_Q08C,width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! queue ! v4l2h264enc capture-io-mode=4
output-io-mode=5 ! queue ! h264parse ! mp4mux ! queue ! filesink
location="/opt/mux2.mp4" camsrc. ! video/x-
raw,format=NV12_Q08C,width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! queue ! v4l2h264enc capture-io-mode=4
output-io-mode=5 ! queue ! h264parse ! mp4mux ! queue ! filesink
location="/opt/mux3.mp4"
```

To stop the use case, use **CTRL + C**. Pull the recorded content out from the device using the following `scp` command and then play the content on host computer.

You can play the MP4 files on a media player. Pull the recorded content using one of the following commands:

```
scp -r root@[DEVICE IP-ADDR]:/opt/mux1.mp4 user1@[HOST IP-ADDR]:/workspace
```

```
scp -r root@[DEVICE IP-ADDR]:/opt/mux2.mp4 user1@[HOST IP-ADDR]:/workspace
```

```
scp -r root@[DEVICE IP-ADDR]:/opt/mux3.mp4 user1@[HOST IP-ADDR]:/workspace
```

The figure shows the flow of the pipeline execution:

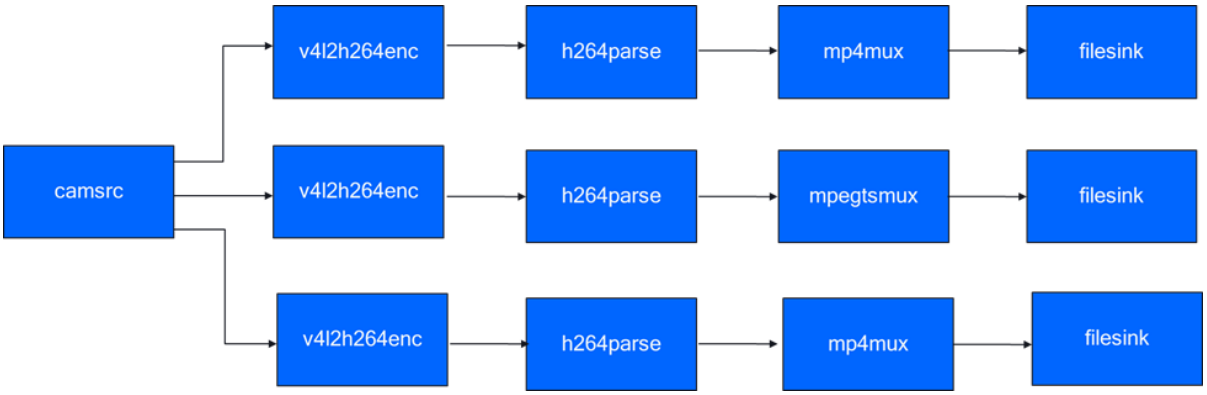


Figure 5-35 Pipeline for encoding three 1080p camera streams

5.2.2.4 Three streams–720p AVC, 720p HEVC, and 720p YUV from live source

The use case uses three 720p streams from the camera. Two of these streams are encoded—one stream encoded as H.264 and the other as HEVC. These streams are multiplexed into different files. The third stream is displayed.

NOTE This use case isn't applicable to QCS8275 and QCS9075.

Run the pipeline:

```
gst-launch-1.0 -e qtiqmmfsrc name=camsrc video_0::type=preview
video_1::type=video ! video/x-
raw,format=NV12_Q08C,width=1280,height=720,framerate=30/1 ! queue !
v4l2h264enc capture-io-mode=4 output-io-mode=5 ! queue ! h264parse ! mp4mux !
queue ! filesink location="/opt/mux1.mp4" camsrc. ! video/x-
raw,format=NV12_Q08C,width=1280,height=720,framerate=30/1 ! queue !
v4l2h265enc capture-io-mode=4 output-io-mode=5 ! queue ! h265parse ! mp4mux !
queue ! filesink location="/opt/mux_hevc.mp4" camsrc. ! video/x-
raw,format=NV12_Q08C,width=1280,height=720,framerate=30/1 ! waylandsink
fullscreen=true
```

To stop the use case, use **CTRL + C**. Pull the recorded content from the device using the following `scp` command and then play the content on host PC.

The camera stream is displayed. You can play the MP4 file on a media player.

```
scp -r root@[DEVICE IP-ADDR]:/opt/mux1.mp4 user1@[HOST IP-ADDR]:/workspace/
```

```
scp -r root@[DEVICE IP-ADDR]:/opt/mux_hevc.mp4 user1@[HOST IP-ADDR]:/
workspace/
```

The figure shows the flow of the pipeline execution:

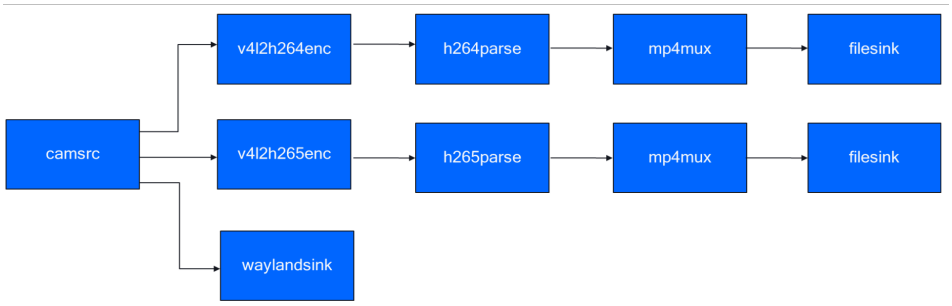


Figure 5-36 Pipeline for encoding 720p AVC, 720p HEVC, and 720p YUV camera streams

5.2.2.5 Three streams–720p AVC MP4, 720p AVC MPEGTS, and 720p AVC MP4 from live source

The use case uses three 720p streams from the camera for encoding. Each stream is encoded as H.264 and multiplexed into different files–two as MP4 and one as MPEGTS.

Run the pipeline:

```
gst-launch-1.0 -e qtiqmmfsrc name=camsrc video_0::type=preview
video_1::type=video ! video/x-
raw,format=NV12_Q08C,width=1280,height=720,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! queue ! v4l2h264enc capture-io-mode=4
output-io-mode=5 ! queue ! h264parse ! mp4mux ! queue ! filesink
location="/opt/mux1.mp4" camsrc. ! video/x-
raw,format=NV12_Q08C,width=1280,height=720,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! queue ! v4l2h264enc capture-io-mode=4
output-io-mode=5 ! queue ! h264parse ! mpegtsmux name=muxer ! queue !
filesink location="/opt/mux_mpegts.mp4" camsrc. ! video/x-
raw,format=NV12_Q08C,width=1280,height=720,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! queue ! v4l2h265enc capture-io-mode=4
output-io-mode=5 ! queue ! h265parse ! mp4mux ! queue ! filesink
location="/opt/mux3.mp4"
```

To stop the use case, use **CTRL + C**.

Pull recorded content from the device using the following `scp` commands and play content on the host computer using a media player.

```
scp -r root@[DEVICE IP-ADDR]:/opt/mux1.mp4 user1@[HOST IP-ADDR]:/workspace
```

```
scp -r root@[DEVICE IP-ADDR]:/opt/mux_mpegts.mp4 user1@[HOST IP-ADDR]:/
workspace
```

```
scp -r root@[DEVICE IP-ADDR]:/opt/mux3.mp4 user1@[HOST IP-ADDR]:/workspace
```

The figure shows the flow of the pipeline execution:

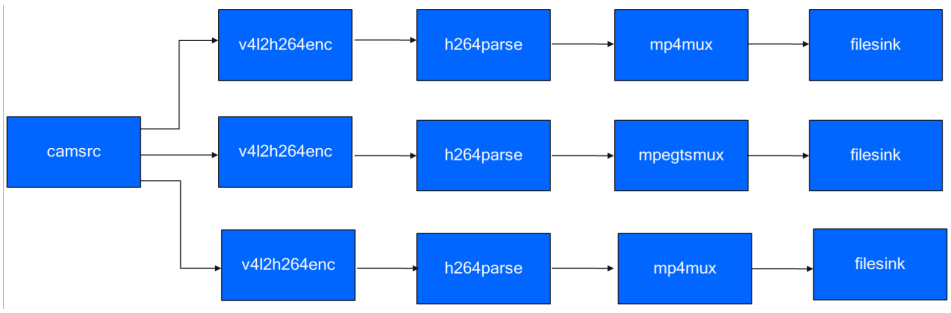


Figure 5-37 Pipeline for encoding 720p AVC MP4, 720p AVC MPEGTS, and 720p AVC MP4 camera streams

5.2.2.6 Three streams–1080p AVC file save, 1080p AVC RTSP, and 1080p YUV from live source

The use case uses three 1080p streams from the camera. Two are encoded as H.264. The first stream is multiplexed and saved to a file. The second stream is sent over the network through RTSP streaming. The third stream is sent to the display.

Run the pipeline:

```
# Run RTSP server in a separate console on target with udpsrc (can be run in
backround as service):
gst-rtsp-server -p 8900 -a <IP address of device>  -m /live "( udpsrc
name=pay0 port=8554 caps=\"application/x-rtp,media=video,clock-
rate=90000,encoding-name=H264,payload=96\" )"
```

```
# Run the pipeline in same or another console on target:
gst-launch-1.0 -e qtiqmmfsrc name=camsrc video_0::type=preview ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! queue ! v4l2h264enc capture-io-mode=4
output-io-mode=5 ! queue ! h264parse ! mp4mux ! queue ! filesink
location="/opt/mux.mp4" camsrc. ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! queue ! v4l2h264enc capture-io-mode=4
output-io-mode=5 ! queue ! h264parse config-interval=-1 ! rtph264pay pt=96 !
udpsink host=<ip_address> port=8554 camsrc. ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1 ! waylandsink
fullscreen=true
```

To stop the use case, use **CTRL + C**.

Pull recorded content from device using the following `scp` command, and play content on host computer.

```
scp -r root@[DEVICE IP-ADDR]:/opt/mux.mp4 user1@[HOST IP-ADDR]:/workspace/
```

You can play the MP4 file on any media player. The streamed video is played as an RTSP stream on a host. The third stream from the camera is displayed.

The figure shows the flow of the pipeline execution:

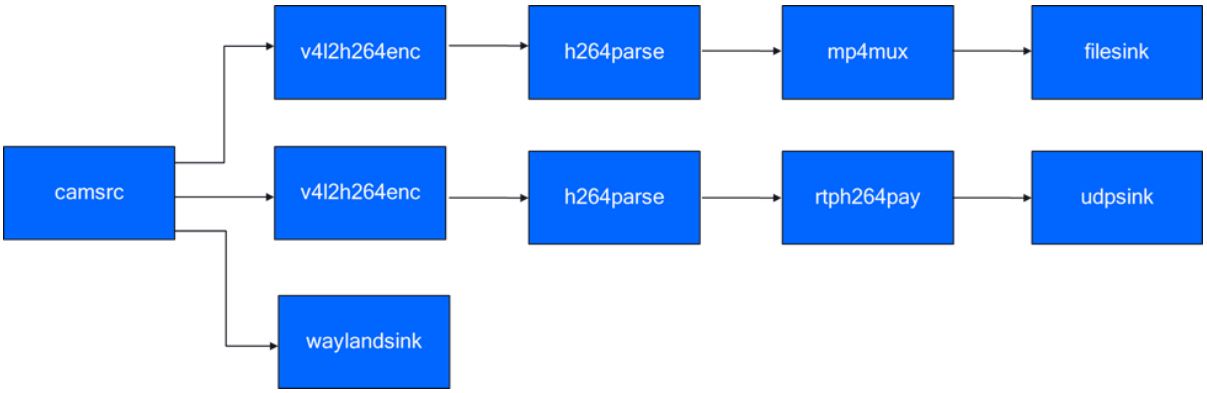


Figure 5-38 Pipeline for encoding three 1080p camera streams

RELATED INFORMATION

"View RTSP stream on host computer" on page 272

5.2.2.7 Three stream–4K JPEG snapshot, 1080p AVC MP4, 1080p YUV from live source

The use case uses three streams from the camera. The first stream is a 4k stream for JPEG snapshot. The second stream, a 1080p stream, is encoded as H.264 and then multiplexed. The third 1080p stream is displayed.

NOTE For QCS9075 and QCS8275, JPEG video encoding is currently not supported.

Run the pipeline:

```
gst-pipeline-app -e qtiqmmfsrc name=camsrc video_0::type=preview ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! queue ! v4l2h264enc capture-io-mode=4
output-io-mode=5 ! queue ! h264parse ! mp4mux ! queue ! filesink
location="/opt/mux.mp4" camsrc.image_1 ! "image/
jpeg,width=3840,height=2160,framerate=30/1" ! multifilesink location=/opt/
frame%d.jpg sync=true async=false camsrc. ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1 ! waylandsink
fullscreen=true
```

A menu is displayed on the `cmd` line terminal.

1. Select **Playing** mode. The pipeline starts the streaming.
2. Select **Enter**.
3. Select the **Plugin** mode and select `camsrc`.
4. Select the **Capture image** option.
5. Select either **video** or **still capture**.
6. Enter the number of snapshots that you want to capture.

To stop the use case, select **Back** and then **Quit** the options from the menu.

You can play the MP4 file and snapshots on a media player. The stream from camera is displayed.

Pull recorded content out from device and play content on the host computer.

```
scp -r root@[DEVICE IP-ADDR]:/opt/mux.mp4 user1@[HOST IP-ADDR]:/workspace
scp -r root@[DEVICE IP-ADDR]:/opt/frame*.jpg user1@[HOST IP-ADDR]:/workspace
```

The figure shows the flow of the pipeline execution:

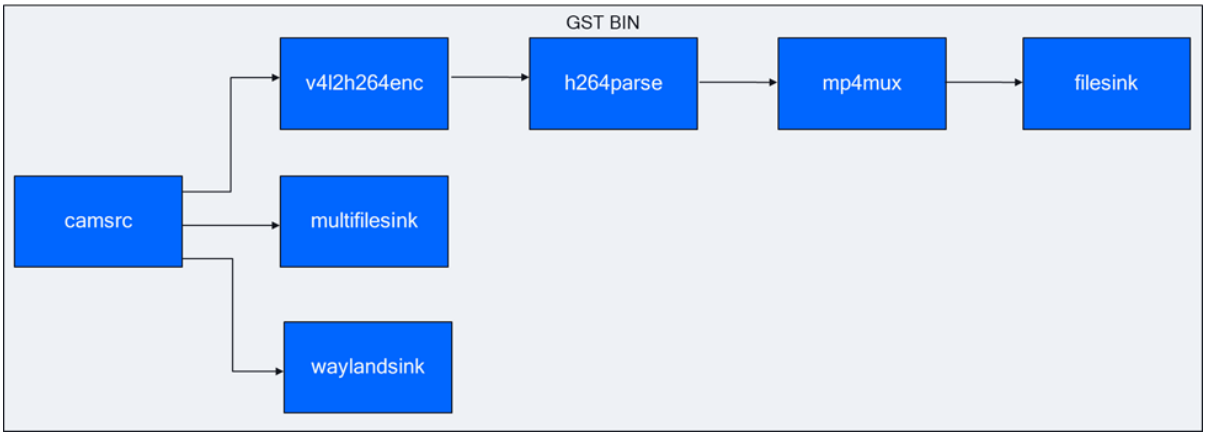


Figure 5-39 Pipeline for three camera streams–4k to JPEG snapshot, 1080p to encoding, 1080 to display

5.2.2.8 Two stream–1080p MJPEG encoder

The MJPEG encoder encodes and writes the captured video stream in an MJPEG format. Run the use cases for encoding, saving to file, and previewing on display.

NOTE For QCS9075 and QCS8275, the MJPEG video encoding is currently not supported.

The following are the various use cases for this encoder. Before running the use cases, type *3* in the terminal to enter Playing state:

- H.264 encode and MJPEG dump to a file:

  ```
  gst-pipeline-app -e qtiqmmfsrc name=camsrc video_0::type=preview ! video/x-raw,format=NV12,width=1920,height=1080,framerate=30/1,interlace-mode=progressive,colorimetry=bt601 ! queue ! v4l2h264enc capture-io-mode=4 output-io-mode=5 ! queue ! h264parse ! mp4mux ! queue ! filesink location="/opt/mux.mp4" camsrc. ! "image/jpeg,width=3840,height=2160,framerate=30/1" ! queue ! avimux ! filesink location=/opt/4k_mjpeg.avi sync=true async=false
  ```

  The `mux.mp4` and `4k_mjpeg.avi` files are stored at `/opt/`.

- H.265 encode and MJPEG dump to a file:

  ```
  gst-pipeline-app -e qtiqmmfsrc name=camsrc video_0::type=preview ! video/x-raw,format=NV12,width=1920,height=1080,framerate=30/1,interlace-mode=progressive,colorimetry=bt601 ! queue ! v4l2h265enc capture-io-mode=4 output-io-mode=5 ! queue ! h265parse ! mp4mux ! queue ! filesink location="/opt/mux.mp4" camsrc. ! "image/jpeg,width=3840,height=2160,framerate=30/1" ! queue ! avimux ! filesink location=/opt/4k_mjpeg.avi sync=true async=false
  ```

  The `mux.mp4` and `4k_mjpeg.avi` files are stored at `/opt/`.

- Preview on display with MJPEG encoding in a file:

```
gst-pipeline-app -e qtiqmmfsrc name=camsrc video_0::type=preview ! video/x-raw,format=NV12,width=1920,height=1080,framerate=30/1,interlace-mode=progressive,colorimetry=bt601 ! queue ! waylandsink fullscreen=true camsrc.video_1 ! "image/jpeg,width=1920,height=1080,framerate=30/1" ! queue ! avimux ! filesink location=/opt/mjpeg.avi sync=true async=false
```

  The video is previewed on a screen and the `mjpeg.avi` file is stored at `/opt/`.

- Preview on display with MJPEG and H.264 encoding in a file:

```
gst-pipeline-app -e qtiqmmfsrc name=camsrc video_0::type=preview ! video/x-raw,format=NV12,width=1920,height=1080,framerate=30/1,interlace-mode=progressive,colorimetry=bt601 ! queue ! waylandsink fullscreen=true camsrc.video_1 ! "image/jpeg,width=1920,height=1080,framerate=30/1" ! queue ! avimux ! filesink location=/opt/mjpeg.avi sync=true async=false camsrc.video_2 ! video/x-raw,format=NV12,width=1920,height=1080,framerate=30/1,interlace-mode=progressive,colorimetry=bt601 ! queue ! v4l2h264enc capture-io-mode=4 output-io-mode=5 ! queue ! h264parse ! mp4mux ! queue ! filesink location="/opt/mux.mp4"
```

  The video is previewed on a screen and the `mux.mp4` and `mjpeg.avi` files are stored at `/opt/`.

5.2.2.9 Slice-based encoding

With slice support for video encoding, the video stream is divided or sliced into several sections so that you can encode them in parallel.

You can use slice-based encoding for video playback, teleconferencing, and security camera use cases.

The following commands enable you to implement slice-based encoding on a camera stream. The output `video.mp4` file is stored at `/opt/`.

H.264

- 1280 × 720, maximum slices per frame = 10:

```
gst-launch-1.0 -e qtiqmmfsrc ! video/x-raw,format=NV12,width=1280,height=720,framerate=30/1,interlace-mode=progressive,colorimetry=bt601 ! queue ! v4l2h264enc capture-io-mode=4 output-io-mode=4 extra-controls="controls,h264_profile=4,h264_level=14,video_bitrate=4800000,video_bitrate_mode=1,frame_level_rate_control_enable=1,video_gop_size=299,h264_entropy_mode=1,slice_partitioning_method=1,numbe_of_mbs_in_a_slice=1200,video_peak_bitrate=32000000;" ! filesink location="/opt/spincalender1280X760_1200_MB_720p_live1800.h264"
```

- 1280 × 720, slices per frame = 8:

```
gst-launch-1.0 -e qtiqmmfsrc ! video/x-raw,format=NV12,width=1280,height=720,framerate=30/1,interlace-mode=progressive,colorimetry=bt601 ! queue ! v4l2h264enc capture-io-mode=4
```

```
output-io-mode=5 extra-
controls="controls,slice_partitioning_method=1,number_of_mbs_in_a_slice=460
;" ! filesink location="/opt/encoded.h264"
```

- 1920 × 1080, slices per frame =3:

```
gst-launch-1.0 -e qtiqmmfsrc ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! queue ! v4l2h264enc capture-io-mode=4
output-io-mode=5 extra-
controls="controls,slice_partitioning_method=1,number_of_mbs_in_a_slice=272
0;" ! filesink location="/opt/encoded.h264"
```

- 1280 × 720, Max Bytes mode:

```
gst-launch-1.0 -e qtiqmmfsrc ! video/x-
raw,format=NV12,width=1280,height=720,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! queue ! v4l2h264enc capture-io-mode=4
output-io-mode=5 extra-controls="controls,slice_partitioning_method=2;" !
filesink location="/opt/encoded.h264"
```

H.265

- 1280 × 720, maximum slices per frame = 10:

```
gst-launch-1.0 -e qtiqmmfsrc ! video/x-
raw,format=NV12,width=1280,height=720,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! queue ! v4l2h265enc capture-io-mode=4
output-io-mode=5 extra-controls="controls,slice_partitioning_method=1;" !
filesink location="/opt/encoded.h265"
```

- 1280 × 720, slices per frame = 8:

```
gst-launch-1.0 -e qtiqmmfsrc ! video/x-
raw,format=NV12,width=1280,height=720,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! queue ! v4l2h265enc capture-io-mode=4
output-io-mode=5 extra-
controls="controls,slice_partitioning_method=1,number_of_mbs_in_a_slice=460
;" ! filesink location="/opt/encoded.h265"
```

- 1920 × 1080, slices per frame = 3:

```
gst-launch-1.0 -e qtiqmmfsrc ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! queue ! v4l2h265enc capture-io-mode=4
output-io-mode=5 extra-
controls="controls,slice_partitioning_method=1,number_of_mbs_in_a_slice=272
0;" ! filesink location="/opt/encoded.h265"
```

- 1280 × 720, Max Bytes mode:

```
gst-launch-1.0 -e qtiqmmfsrc ! video/x-
raw,format=NV12,width=1280,height=720,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! queue ! v4l2h265enc capture-io-mode=4
output-io-mode=5 extra-controls="controls,slice_partitioning_method=2;" !
filesink location="/opt/encoded.h265"
```

5.2.2.10 Intra-frame smart codec (H.264 and H.265)

Intra-frame (I frame) is a data compression technique that allows smaller file sizes and lower bitrates. Run the use cases for the 720p@30 HEVC streams with smart codec and noise reduction.

For the following use cases, the output `video.mp4` file is stored at `/opt/`.

720p@30 HEVC stream with smart bitrate, smart framerate, and smart GOP–noise reduction enabled

- H.264

```
gst-launch-1.0 -ev qtiqmmfsrc noise-reduction=2 video_0::extra-buffers=20
video_0::type=preview name=camsrc ! video/x-
raw,format=NV12,width=1280,height=720,framerate=30/1 ! queue ! scb.sink
qtismartvencbin default-gop=30 max-gop=600 levels-
override="LevelsOverride,bitrate_static=160000,bitrate_low=358000,bitrate_m
edium=700000,bitrate_high=1400000,fr_static=16,fr_low=4,fr_medium=2,fr_high
=1;" roi-quality-cfg="ROIQPs,car=2,person=1,tree=-2;"
encoder="v4l2h264enc" max-bitrate=4200000 name=scb ! queue ! h264parse !
queue ! mp4mux ! queue ! filesink location=/opt/video.mp4 camsrc. !
video/x-raw,format=NV12,width=640,height=480,framerate=15/1 ! queue !
scb.sink_ctrl
```

- H.265

```
gst-launch-1.0 -ev qtiqmmfsrc noise-reduction=2 video_0::extra-buffers=20
video_0::type=preview name=camsrc ! video/x-
raw,format=NV12,width=1280,height=720,framerate=30/1 ! queue ! scb.sink
qtismartvencbin default-gop=30 max-gop=600 levels-
override="LevelsOverride,bitrate_static=160000,bitrate_low=358000,bitrate_m
edium=700000,bitrate_high=1400000,fr_static=16,fr_low=4,fr_medium=2,fr_high
=1;" roi-quality-cfg="ROIQPs,car=2,person=1,tree=-2;"
encoder="v4l2h265enc" max-bitrate=4200000 name=scb ! queue ! h265parse !
queue ! mp4mux ! queue ! filesink location=/opt/video.mp4 camsrc. !
video/x-raw,format=NV12,width=640,height=480,framerate=15/1 ! queue !
scb.sink_ctrl
```

720p@30 HEVC stream with smart bitrate only–noise reduction enabled

- H.264

```
gst-launch-1.0 -ev qtiqmmfsrc noise-reduction=2 video_0::extra-buffers=20
video_0::type=preview name=camsrc ! video/x-
raw,format=NV12,width=1280,height=720,framerate=30/1 ! queue ! scb.sink
qtismartvencbin default-gop=30 max-gop=600 smart-framerate=false smart-
gop=false encoder="v4l2h264enc" max-bitrate=4200000 name=scb ! queue !
h264parse ! queue ! mp4mux ! queue ! filesink location=/opt/video.mp4
camsrc. ! video/x-raw,format=NV12,width=640,height=480,framerate=15/1 !
queue ! scb.sink_ctrl
```

- H.265

```
gst-launch-1.0 -ev qtiqmmfsrc noise-reduction=2 video_0::extra-buffers=20
video_0::type=preview name=camsrc ! video/x-
raw,format=NV12,width=1280,height=720,framerate=30/1 ! queue ! scb.sink
qtismartvencbin default-gop=30 max-gop=600 smart-framerate=false smart-
gop=false encoder="v4l2h265enc" max-bitrate=4200000 name=scb ! queue !
h265parse ! queue ! mp4mux ! queue ! filesink location=/opt/video.mp4
camsrc. ! video/x-raw,format=NV12,width=640,height=480,framerate=15/1 !
queue ! scb.sink_ctrl
```

720p@30 HEVC stream with smart bitrate, smart framerate, and smart GOP–noise reduction disabled

- H.264

```
gst-launch-1.0 -ev qtiqmmfsrc video_0::extra-buffers=20
video_0::type=preview name=camsrc ! video/x-
raw,format=NV12,width=1280,height=720,framerate=30/1 ! queue ! scb.sink
qtismartvencbin default-gop=30 max-gop=600 encoder="v4l2h264enc" max-
bitrate=4200000 name=scb ! queue ! h264parse ! queue ! mp4mux ! queue !
filesink location=/opt/video.mp4 camsrc. ! video/x-
raw,format=NV12,width=640,height=480,framerate=15/1 ! queue ! scb.sink_ctrl
```

- H.265

```
gst-launch-1.0 -ev qtiqmmfsrc video_0::extra-buffers=20
video_0::type=preview name=camsrc ! video/x-
raw,format=NV12,width=1280,height=720,framerate=30/1 ! queue ! scb.sink
qtismartvencbin default-gop=30 max-gop=600 encoder="v4l2h265enc" max-
bitrate=4200000 name=scb ! queue ! h265parse ! queue ! mp4mux ! queue !
filesink location=/opt/video.mp4 camsrc. ! video/x-
raw,format=NV12,width=640,height=480,framerate=15/1 ! queue ! scb.sink_ctrl
```

720p@30 HEVC streamed over TCP with smart bitrate, smart framerate, and smart GOP–noise reduction enabled

- H.264

```
gst-launch-1.0 -ev qtiqmmfsrc noise-reduction=2 video_0::extra-buffers=20
video_0::type=preview name=camsrc ! video/x-
raw,format=NV12,width=1280,height=720,framerate=30/1 ! queue ! scb.sink
qtismartvencbin default-gop=30 max-gop=600 encoder="v4l2h264enc" max-
bitrate=4200000 name=scb ! queue ! h264parse config-interval=1 ! mpegtsmux
name=muxer ! queue ! tcpserversink port=8900 host=<ip_address> camsrc. !
video/x-raw,format=NV12,width=640,height=480,framerate=15/1 ! queue !
scb.sink_ctrl
```

- H.265

```
gst-launch-1.0 -ev qtiqmmfsrc noise-reduction=2 video_0::extra-buffers=20
video_0::type=preview name=camsrc ! video/x-
raw,format=NV12,width=1280,height=720,framerate=30/1 ! queue ! scb.sink
```

```
qtismartvencbin default-gop=30 max-gop=600 encoder="v4l2h265enc" max-
bitrate=4200000 name=scb ! queue ! h265parse config-interval=1 ! mpegtsmux
name=muxer ! queue ! tcpserversink port=8900 host=<ip_address> camsrc. !
video/x-raw,format=NV12,width=640,height=480,framerate=15/1 ! queue !
scb.sink_ctrl
```

To play the stream over TCP, do the following on the Windows host:

1. Open the VLC media player.
2. Select **Media** > **Open Network Stream** or use **CTRL + N**).
3. Enter `tcp://<ip_address>:8900/live`.
4. Select **Play**.

720p@30 HEVC stream with smart bitrate, smart framerate, and smart GOP–noise reduction enabled

- H.264

```
gst-launch-1.0 -ev qtiqmmfsrc noise-reduction=2 video_0::extra-buffers=20
video_0::type=preview name=camsrc ! video/x-
raw,format=NV12,width=1280,height=720,framerate=30/1 ! queue ! scb.sink
qtismartvencbin default-gop=30 max-gop=600 encoder="v4l2h264enc" max-
bitrate=1000000 name=scb ! queue ! h264parse ! queue ! mp4mux ! queue !
filesink location=/opt/video.mp4 camsrc. ! video/x-
raw,format=NV12,width=640,height=480,framerate=15/1 ! queue ! tee
name=split ! queue ! scb.sink_ctrl split. ! queue ! qtimlvconverter !
queue ! qtimltflite delegate=external external-delegate-
path=libQnnTFLiteDelegate.so external-delegate-
options="QNNExternalDelegate,backend_type=htp;" model=<model-path> !
queue ! qtimlvdetection threshold=50.0 results=5 module=yolov8
labels=<labels-path> constants="YOLOv8,q-offsets=<-107.0, -128.0, 0.0>,q-
scales=<3.093529462814331, 0.00390625, 1.0>;" ! text/x-raw ! queue !
scb.sink_ml
```

- H.265

```
gst-launch-1.0 -ev qtiqmmfsrc noise-reduction=2 video_0::extra-buffers=20
video_0::type=preview name=camsrc ! video/x-
raw,format=NV12,width=1280,height=720,framerate=30/1 ! queue ! scb.sink
qtismartvencbin default-gop=30 max-gop=600 encoder="v4l2h265enc" max-
bitrate=1000000 name=scb ! queue ! h265parse ! queue ! mp4mux ! queue !
filesink location=/opt/video.mp4 camsrc. ! video/x-
raw,format=NV12,width=640,height=480,framerate=15/1 ! queue ! tee
name=split ! queue ! scb.sink_ctrl split. ! queue ! qtimlvconverter !
queue ! qtimltflite delegate=external external-delegate-
path=libQnnTFLiteDelegate.so external-delegate-
options="QNNExternalDelegate,backend_type=htp;" model=<model-path> !
queue ! qtimlvdetection threshold=50.0 results=5 module=yolov8
labels=<labels-path> constants="YOLOv8,q-offsets=<-107.0, -128.0, 0.0>,q-
```

```
scales=<3.093529462814331, 0.00390625, 1.0>;" ! text/x-raw ! queue !
scb.sink_ml
```

Concurrent two 720p@30 HEVC streams with smart bitrate, smart framerate,and smart GOP–noise reduction enabled

- H.264

```
gst-launch-1.0 -ev qtiqmmfsrc noise-reduction=2 video_0::extra-buffers=20
video_0::type=preview video_1::extra-buffers=20 name=camsrc ! video/x-
raw,format=NV12,width=1280,height=720,framerate=30/1 ! queue ! scb1.sink
qtismartvencbin default-gop=30 max-gop=600 encoder="v4l2h264enc" max-
bitrate=4200000 name=scb1 ! queue ! h264parse ! queue ! mp4mux ! queue !
filesink location=/opt/video_scb1.mp4 camsrc. ! video/x-
raw,format=NV12,width=1280,height=720,framerate=30/1 ! queue ! scb2.sink
qtismartvencbin default-gop=30 max-gop=600 encoder="v4l2h264enc" max-
bitrate=4200000 name=scb2 ! queue ! h264parse ! queue ! mp4mux ! queue !
filesink location=/opt/video_scb2.mp4 camsrc. ! video/x-
raw,format=NV12,width=640,height=480,framerate=15/1 ! queue ! tee
name=ctrl_t ! queue ! scb1.sink_ctrl ctrl_t. ! queue ! scb2.sink_ctrl
```

- H.265

```
gst-launch-1.0 -ev qtiqmmfsrc noise-reduction=2 video_0::extra-buffers=20
video_0::type=preview video_1::extra-buffers=20 name=camsrc ! video/x-
raw,format=NV12,width=1280,height=720,framerate=30/1 ! queue ! scb1.sink
qtismartvencbin default-gop=30 max-gop=600 encoder="v4l2h265enc" max-
bitrate=4200000 name=scb1 ! queue ! h265parse ! queue ! mp4mux ! queue !
filesink location=/opt/video_scb1.mp4 camsrc. ! video/x-
raw,format=NV12,width=1280,height=720,framerate=30/1 ! queue ! scb2.sink
qtismartvencbin default-gop=30 max-gop=600 encoder="v4l2h265enc" max-
bitrate=4200000 name=scb2 ! queue ! h265parse ! queue ! mp4mux ! queue !
filesink location=/opt/video_scb2.mp4 camsrc. ! video/x-
raw,format=NV12,width=640,height=480,framerate=15/1 ! queue ! tee
name=ctrl_t ! queue ! scb1.sink_ctrl ctrl_t. ! queue ! scb2.sink_ctrl
```

5.2.2.11 MJPEG video encode in CPU and GPU

For MJPEG encoding, if the hardware isn't available, support is required for CPU and GPU. The use cases include the capabilities to encode a 16-channel 1080p@2fps video.

NOTE For QCS9075 and QCS8275, the MJPEG video encoding is currently not supported.

Prerequisites

For MJPEG encoding, support for 1080p@2fps in CPU/GPU is required.

Set the following environment variables before running the use cases:

```
export XDG_RUNTIME_DIR=/dev/socket/weston
export WAYLAND_DISPLAY=wayland-1
```

YUV file 1920 × 1080@30 → 16 × JPEG-ENC → AVI file

```
gst-launch-1.0 -e filesrc location=/opt/1920_1080.yuv ! videoparse width=1920
height=1080 format=nv12 framerate=2/1 ! jpegenc ! avimux ! filesink
location=/opt/mjpeg1.avi &
gst-launch-1.0 -e filesrc location=/opt/1920_1080.yuv ! videoparse width=1920
height=1080 format=nv12 framerate=2/1 ! jpegenc ! avimux ! filesink
location=/opt/mjpeg2.avi &
gst-launch-1.0 -e filesrc location=/opt/1920_1080.yuv ! videoparse width=1920
height=1080 format=nv12 framerate=2/1 ! jpegenc ! avimux ! filesink
location=/opt/mjpeg3.avi &
gst-launch-1.0 -e filesrc location=/opt/1920_1080.yuv ! videoparse width=1920
height=1080 format=nv12 framerate=2/1 ! jpegenc ! avimux ! filesink
location=/opt/mjpeg4.avi &
gst-launch-1.0 -e filesrc location=/opt/1920_1080.yuv ! videoparse width=1920
height=1080 format=nv12 framerate=2/1 ! jpegenc ! avimux ! filesink
location=/opt/mjpeg5.avi &
gst-launch-1.0 -e filesrc location=/opt/1920_1080.yuv ! videoparse width=1920
height=1080 format=nv12 framerate=2/1 ! jpegenc ! avimux ! filesink
location=/opt/mjpeg6.avi &
gst-launch-1.0 -e filesrc location=/opt/1920_1080.yuv ! videoparse width=1920
height=1080 format=nv12 framerate=2/1 ! jpegenc ! avimux ! filesink
location=/opt/mjpeg7.avi &
gst-launch-1.0 -e filesrc location=/opt/1920_1080.yuv ! videoparse width=1920
height=1080 format=nv12 framerate=2/1 ! jpegenc ! avimux ! filesink
location=/opt/mjpeg8.avi &
gst-launch-1.0 -e filesrc location=/opt/1920_1080.yuv ! videoparse width=1920
height=1080 format=nv12 framerate=2/1 ! jpegenc ! avimux ! filesink
location=/opt/mjpeg9.avi &
gst-launch-1.0 -e filesrc location=/opt/1920_1080.yuv ! videoparse width=1920
height=1080 format=nv12 framerate=2/1 ! jpegenc ! avimux ! filesink
location=/opt/mjpeg10.avi &
gst-launch-1.0 -e filesrc location=/opt/1920_1080.yuv ! videoparse width=1920
height=1080 format=nv12 framerate=2/1 ! jpegenc ! avimux ! filesink
location=/opt/mjpeg11.avi &
gst-launch-1.0 -e filesrc location=/opt/1920_1080.yuv ! videoparse width=1920
height=1080 format=nv12 framerate=2/1 ! jpegenc ! avimux ! filesink
location=/opt/mjpeg12.avi &
gst-launch-1.0 -e filesrc location=/opt/1920_1080.yuv ! videoparse width=1920
height=1080 format=nv12 framerate=2/1 ! jpegenc ! avimux ! filesink
location=/opt/mjpeg13.avi &
gst-launch-1.0 -e filesrc location=/opt/1920_1080.yuv ! videoparse width=1920
height=1080 format=nv12 framerate=2/1 ! jpegenc ! avimux ! filesink
location=/opt/mjpeg14.avi &
gst-launch-1.0 -e filesrc location=/opt/1920_1080.yuv ! videoparse width=1920
height=1080 format=nv12 framerate=2/1 ! jpegenc ! avimux ! filesink
location=/opt/mjpeg15.avi &
```

```
gst-launch-1.0 -e filesrc location=/opt/1920_1080.yuv ! videoparse width=1920
height=1080 format=nv12 framerate=2/1 ! jpegenc ! avimux ! filesink
location=/opt/mjpeg16.avi
```

YUV file 1920 × 1080@30 → 12 × JPEG-ENC → AVI file and 1920 × 1080@30 AVI → 12 × JPEG-DEC > JPEG

```
gst-launch-1.0 -e filesrc location=/opt/mjpeg.avi ! avidemux !  queue !
jpegdec ! multifilesink location=/opt/mjpeg1_%d.jpeg max-files=1 &
gst-launch-1.0 -e filesrc location=/opt/mjpeg.avi ! avidemux !  queue !
jpegdec ! multifilesink location=/opt/mjpeg2_%d.jpeg max-files=1 &
gst-launch-1.0 -e filesrc location=/opt/mjpeg.avi ! avidemux !  queue !
jpegdec ! multifilesink location=/opt/mjpeg3_%d.jpeg max-files=1 &
gst-launch-1.0 -e filesrc location=/opt/mjpeg.avi ! avidemux !  queue !
jpegdec ! multifilesink location=/opt/mjpeg4_%d.jpeg max-files=1 &
gst-launch-1.0 -e filesrc location=/opt/mjpeg.avi ! avidemux !  queue !
jpegdec ! multifilesink location=/opt/mjpeg5_%d.jpeg max-files=1 &
gst-launch-1.0 -e filesrc location=/opt/mjpeg.avi ! avidemux !  queue !
jpegdec ! multifilesink location=/opt/mjpeg6_%d.jpeg max-files=1 &
gst-launch-1.0 -e filesrc location=/opt/mjpeg.avi ! avidemux !  queue !
jpegdec ! multifilesink location=/opt/mjpeg7_%d.jpeg max-files=1 &
gst-launch-1.0 -e filesrc location=/opt/mjpeg.avi ! avidemux !  queue !
jpegdec ! multifilesink location=/opt/mjpeg8_%d.jpeg max-files=1 &
gst-launch-1.0 -e filesrc location=/opt/mjpeg.avi ! avidemux !  queue !
jpegdec ! multifilesink location=/opt/mjpeg9_%d.jpeg max-files=1 &
gst-launch-1.0 -e filesrc location=/opt/mjpeg.avi ! avidemux !  queue !
jpegdec ! multifilesink location=/opt/mjpeg10_%d.jpeg max-files=1 &
gst-launch-1.0 -e filesrc location=/opt/mjpeg.avi ! avidemux !  queue !
jpegdec ! multifilesink location=/opt/mjpeg11_%d.jpeg max-files=1 &
gst-launch-1.0 -e filesrc location=/opt/mjpeg.avi ! avidemux !  queue !
jpegdec ! multifilesink location=/opt/mjpeg12_%d.jpeg max-files=1 &
gst-launch-1.0 -e filesrc location=/opt/1920_1080.yuv ! videoparse width=1920
height=1080 format=nv12 framerate=2/1 ! jpegenc ! avimux ! filesink
location=/opt/mjpeg1.avi &
gst-launch-1.0 -e filesrc location=/opt/1920_1080.yuv ! videoparse width=1920
height=1080 format=nv12 framerate=2/1 ! jpegenc ! avimux ! filesink
location=/opt/mjpeg2.avi &
gst-launch-1.0 -e filesrc location=/opt/1920_1080.yuv ! videoparse width=1920
height=1080 format=nv12 framerate=2/1 ! jpegenc ! avimux ! filesink
location=/opt/mjpeg3.avi &
gst-launch-1.0 -e filesrc location=/opt/1920_1080.yuv ! videoparse width=1920
height=1080 format=nv12 framerate=2/1 ! jpegenc ! avimux ! filesink
location=/opt/mjpeg4.avi &
gst-launch-1.0 -e filesrc location=/opt/1920_1080.yuv ! videoparse width=1920
height=1080 format=nv12 framerate=2/1 ! jpegenc ! avimux ! filesink
location=/opt/mjpeg5.avi &
gst-launch-1.0 -e filesrc location=/opt/1920_1080.yuv ! videoparse width=1920
height=1080 format=nv12 framerate=2/1 ! jpegenc ! avimux ! filesink
```

```
location=/opt/mjpeg6.avi &
gst-launch-1.0 -e filesrc location=/opt/1920_1080.yuv ! videoparse width=1920
height=1080 format=nv12 framerate=2/1 ! jpegenc ! avimux ! filesink
location=/opt/mjpeg7.avi &
gst-launch-1.0 -e filesrc location=/opt/1920_1080.yuv ! videoparse width=1920
height=1080 format=nv12 framerate=2/1 ! jpegenc ! avimux ! filesink
location=/opt/mjpeg8.avi &
gst-launch-1.0 -e filesrc location=/opt/1920_1080.yuv ! videoparse width=1920
height=1080 format=nv12 framerate=2/1 ! jpegenc ! avimux ! filesink
location=/opt/mjpeg9.avi &
gst-launch-1.0 -e filesrc location=/opt/1920_1080.yuv ! videoparse width=1920
height=1080 format=nv12 framerate=2/1 ! jpegenc ! avimux ! filesink
location=/opt/mjpeg10.avi &
gst-launch-1.0 -e filesrc location=/opt/1920_1080.yuv ! videoparse width=1920
height=1080 format=nv12 framerate=2/1 ! jpegenc ! avimux ! filesink
location=/opt/mjpeg11.avi &
gst-launch-1.0 -e filesrc location=/opt/1920_1080.yuv ! videoparse width=1920
height=1080 format=nv12 framerate=2/1 ! jpegenc ! avimux ! filesink
location=/opt/mjpeg12.avi
```

5.2.2.12 Enable 24 end-to-end concurrent video sessions

Run the use cases for 24 parallel 1080p streams taken from files and sent to a display. Each stream is displayed at different positions on the screen.

NOTE These use cases aren't supported on QCS6490 and QCS8275.

H.264–24 session decode

```
ulimit -n 4096 && gst-launch-1.0 -e qtivcomposer name=mix
sink_0::position="<0, 0>" sink_0::dimensions="<240, 270>" \
sink_1::position="<240, 0>" sink_1::dimensions="<240, 270>" \
sink_2::position="<480, 0>" sink_2::dimensions="<240, 270>" \
sink_3::position="<720, 0>" sink_3::dimensions="<240, 270>" \
sink_4::position="<960, 0>" sink_4::dimensions="<240, 270>" \
sink_5::position="<1200, 0>" sink_5::dimensions="<240, 270>" \
sink_6::position="<1440, 0>" sink_6::dimensions="<240, 270>" \
sink_7::position="<1680, 0>" sink_7::dimensions="<240, 270>" \
sink_8::position="<0, 270>" sink_8::dimensions="<240, 270>" \
sink_9::position="<240, 270>" sink_9::dimensions="<240, 270>" \
sink_10::position="<480, 270>" sink_10::dimensions="<240, 270>" \
sink_11::position="<720, 270>" sink_11::dimensions="<240, 270>" \
sink_12::position="<960, 270>" sink_12::dimensions="<240, 270>" \
sink_13::position="<1200, 270>" sink_13::dimensions="<240, 270>" \
sink_14::position="<1440, 270>" sink_14::dimensions="<240, 270>" \
sink_15::position="<1680, 270>" sink_15::dimensions="<240, 270>" \
sink_16::position="<0, 540>" sink_16::dimensions="<240, 270>" \
sink_17::position="<240, 540>" sink_17::dimensions="<240, 270>" \
sink_18::position="<480, 540>" sink_18::dimensions="<240, 270>" \
```

```
sink_19::position="<720, 540>" sink_19::dimensions="<240, 270>" \
sink_20::position="<960, 540>" sink_20::dimensions="<240, 270>" \
sink_21::position="<1200, 540>" sink_21::dimensions="<240, 270>" \
sink_22::position="<1440, 540>" sink_22::dimensions="<240, 270>" \
sink_23::position="<1680, 540>" sink_23::dimensions="<240, 270>" \
mix. ! fpsdisplaysink text-overlay=true video-sink="waylandsink
fullscreen=true" -v \
filesrc location=/opt/1920_1080_h264_30fps_1min.mp4 ! qtdemux ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw, format=NV12 !
mix. \
filesrc location=/opt/1920_1080_h264_30fps_1min.mp4 ! qtdemux ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw, format=NV12 !
mix. \
filesrc location=/opt/1920_1080_h264_30fps_1min.mp4 ! qtdemux ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw, format=NV12 !
mix. \
filesrc location=/opt/1920_1080_h264_30fps_1min.mp4 ! qtdemux ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw, format=NV12 !
mix. \
filesrc location=/opt/1920_1080_h264_30fps_1min.mp4 ! qtdemux ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw, format=NV12 !
mix. \
filesrc location=/opt/1920_1080_h264_30fps_1min.mp4 ! qtdemux ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw, format=NV12 !
mix. \
filesrc location=/opt/1920_1080_h264_30fps_1min.mp4 ! qtdemux ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw, format=NV12 !
mix. \
filesrc location=/opt/1920_1080_h264_30fps_1min.mp4 ! qtdemux ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw, format=NV12 !
mix. \
filesrc location=/opt/1920_1080_h264_30fps_1min.mp4 ! qtdemux ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw, format=NV12 !
mix. \
filesrc location=/opt/1920_1080_h264_30fps_1min.mp4 ! qtdemux ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw, format=NV12 !
mix. \
filesrc location=/opt/1920_1080_h264_30fps_1min.mp4 ! qtdemux ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw, format=NV12 !
mix. \
filesrc location=/opt/1920_1080_h264_30fps_1min.mp4 ! qtdemux ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw, format=NV12 !
mix. \
filesrc location=/opt/1920_1080_h264_30fps_1min.mp4 ! qtdemux ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw, format=NV12 !
mix. \
filesrc location=/opt/1920_1080_h264_30fps_1min.mp4 ! qtdemux ! h264parse !
```

```
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw, format=NV12 !
mix. \
filesrc location=/opt/1920_1080_h264_30fps_1min.mp4 ! qtdemux ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw, format=NV12 !
mix. \
filesrc location=/opt/1920_1080_h264_30fps_1min.mp4 ! qtdemux ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw, format=NV12 !
mix. \
filesrc location=/opt/1920_1080_h264_30fps_1min.mp4 ! qtdemux ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw, format=NV12 !
mix. \
filesrc location=/opt/1920_1080_h264_30fps_1min.mp4 ! qtdemux ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw, format=NV12 !
mix. \
filesrc location=/opt/1920_1080_h264_30fps_1min.mp4 ! qtdemux ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw, format=NV12 !
mix. \
filesrc location=/opt/1920_1080_h264_30fps_1min.mp4 ! qtdemux ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw, format=NV12 !
mix. \
filesrc location=/opt/1920_1080_h264_30fps_1min.mp4 ! qtdemux ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw, format=NV12 !
mix. \
filesrc location=/opt/1920_1080_h264_30fps_1min.mp4 ! qtdemux ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw, format=NV12 !
mix. \
filesrc location=/opt/1920_1080_h264_30fps_1min.mp4 ! qtdemux ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw, format=NV12 !
mix. \
filesrc location=/opt/1920_1080_h264_30fps_1min.mp4 ! qtdemux ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw, format=NV12 !
mix. \
filesrc location=/opt/1920_1080_h264_30fps_1min.mp4 ! qtdemux ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw, format=NV12 !
mix.
```

H.265–24 session decode

```
ulimit -n 4096 && gst-launch-1.0 -e qtivcomposer name=mix
sink_0::position="<0, 0>" sink_0::dimensions="<240, 270>" \
sink_1::position="<240, 0>" sink_1::dimensions="<240, 270>" \
sink_2::position="<480, 0>" sink_2::dimensions="<240, 270>" \
sink_3::position="<720, 0>" sink_3::dimensions="<240, 270>" \
sink_4::position="<960, 0>" sink_4::dimensions="<240, 270>" \
sink_5::position="<1200, 0>" sink_5::dimensions="<240, 270>" \
sink_6::position="<1440, 0>" sink_6::dimensions="<240, 270>" \
sink_7::position="<1680, 0>" sink_7::dimensions="<240, 270>" \
sink_8::position="<0, 270>" sink_8::dimensions="<240, 270>" \
sink_9::position="<240, 270>" sink_9::dimensions="<240, 270>" \
sink_10::position="<480, 270>" sink_10::dimensions="<240, 270>" \
sink_11::position="<720, 270>" sink_11::dimensions="<240, 270>" \
```

```
sink_12::position="<960, 270>" sink_12::dimensions="<240, 270>" \
sink_13::position="<1200, 270>" sink_13::dimensions="<240, 270>" \
sink_14::position="<1440, 270>" sink_14::dimensions="<240, 270>" \
sink_15::position="<1680, 270>" sink_15::dimensions="<240, 270>" \
sink_16::position="<0, 540>" sink_16::dimensions="<240, 270>" \
sink_17::position="<240, 540>" sink_17::dimensions="<240, 270>" \
sink_18::position="<480, 540>" sink_18::dimensions="<240, 270>" \
sink_19::position="<720, 540>" sink_19::dimensions="<240, 270>" \
sink_20::position="<960, 540>" sink_20::dimensions="<240, 270>" \
sink_21::position="<1200, 540>" sink_21::dimensions="<240, 270>" \
sink_22::position="<1440, 540>" sink_22::dimensions="<240, 270>" \
sink_23::position="<1680, 540>" sink_23::dimensions="<240, 270>" \
mix. ! fpsdisplaysink text-overlay=false video-sink="waylandsink
fullscreen=true" -v \
filesrc location=/opt/
FlyingBirdFlock_1920x1080_30fps_HEVC_7862Kbps_withAudio.mp4 ! qtdemux !
h265parse ! v4l2h265dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! mix. \
filesrc location=/opt/
FlyingBirdFlock_1920x1080_30fps_HEVC_7862Kbps_withAudio.mp4 ! qtdemux !
h265parse ! v4l2h265dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! mix. \
filesrc location=/opt/
FlyingBirdFlock_1920x1080_30fps_HEVC_7862Kbps_withAudio.mp4 ! qtdemux !
h265parse ! v4l2h265dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! mix. \
filesrc location=/opt/
FlyingBirdFlock_1920x1080_30fps_HEVC_7862Kbps_withAudio.mp4 ! qtdemux !
h265parse ! v4l2h265dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! mix. \
filesrc location=/opt/
FlyingBirdFlock_1920x1080_30fps_HEVC_7862Kbps_withAudio.mp4 ! qtdemux !
h265parse ! v4l2h265dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! mix. \
filesrc location=/opt/
FlyingBirdFlock_1920x1080_30fps_HEVC_7862Kbps_withAudio.mp4 ! qtdemux !
h265parse ! v4l2h265dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! mix. \
filesrc location=/opt/
FlyingBirdFlock_1920x1080_30fps_HEVC_7862Kbps_withAudio.mp4 ! qtdemux !
h265parse ! v4l2h265dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! mix. \
filesrc location=/opt/
FlyingBirdFlock_1920x1080_30fps_HEVC_7862Kbps_withAudio.mp4 ! qtdemux !
h265parse ! v4l2h265dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! mix. \
filesrc location=/opt/
```

```
FlyingBirdFlock_1920x1080_30fps_HEVC_7862Kbps_withAudio.mp4 ! qtdemux !
h265parse ! v4l2h265dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! mix. \
filesrc location=/opt/
FlyingBirdFlock_1920x1080_30fps_HEVC_7862Kbps_withAudio.mp4 ! qtdemux !
h265parse ! v4l2h265dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! mix. \
filesrc location=/opt/
FlyingBirdFlock_1920x1080_30fps_HEVC_7862Kbps_withAudio.mp4 ! qtdemux !
h265parse ! v4l2h265dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! mix. \
filesrc location=/opt/
FlyingBirdFlock_1920x1080_30fps_HEVC_7862Kbps_withAudio.mp4 ! qtdemux !
h265parse ! v4l2h265dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! mix. \
filesrc location=/opt/
FlyingBirdFlock_1920x1080_30fps_HEVC_7862Kbps_withAudio.mp4 ! qtdemux !
h265parse ! v4l2h265dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! mix. \
filesrc location=/opt/
FlyingBirdFlock_1920x1080_30fps_HEVC_7862Kbps_withAudio.mp4 ! qtdemux !
h265parse ! v4l2h265dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! mix. \
filesrc location=/opt/
FlyingBirdFlock_1920x1080_30fps_HEVC_7862Kbps_withAudio.mp4 ! qtdemux !
h265parse ! v4l2h265dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! mix. \
filesrc location=/opt/
FlyingBirdFlock_1920x1080_30fps_HEVC_7862Kbps_withAudio.mp4 ! qtdemux !
h265parse ! v4l2h265dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! mix. \
filesrc location=/opt/
FlyingBirdFlock_1920x1080_30fps_HEVC_7862Kbps_withAudio.mp4 ! qtdemux !
h265parse ! v4l2h265dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! mix. \
filesrc location=/opt/
FlyingBirdFlock_1920x1080_30fps_HEVC_7862Kbps_withAudio.mp4 ! qtdemux !
h265parse ! v4l2h265dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! mix. \
filesrc location=/opt/
FlyingBirdFlock_1920x1080_30fps_HEVC_7862Kbps_withAudio.mp4 ! qtdemux !
h265parse ! v4l2h265dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! mix. \
filesrc location=/opt/
FlyingBirdFlock_1920x1080_30fps_HEVC_7862Kbps_withAudio.mp4 ! qtdemux !
h265parse ! v4l2h265dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! mix. \
filesrc location=/opt/
FlyingBirdFlock_1920x1080_30fps_HEVC_7862Kbps_withAudio.mp4 ! qtdemux !
h265parse ! v4l2h265dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! mix. \
```

```
filesrc location=/opt/
FlyingBirdFlock_1920x1080_30fps_HEVC_7862Kbps_withAudio.mp4 ! qtdemux !
h265parse ! v4l2h265dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! mix. \
filesrc location=/opt/
FlyingBirdFlock_1920x1080_30fps_HEVC_7862Kbps_withAudio.mp4 ! qtdemux !
h265parse ! v4l2h265dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! mix. \
filesrc location=/opt/
FlyingBirdFlock_1920x1080_30fps_HEVC_7862Kbps_withAudio.mp4 ! qtdemux !
h265parse ! v4l2h265dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! mix. \
filesrc location=/opt/
FlyingBirdFlock_1920x1080_30fps_HEVC_7862Kbps_withAudio.mp4 ! qtdemux !
h265parse ! v4l2h265dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! mix.
```

AV1–24 session decode

```
ulimit -n 4096 && gst-launch-1.0 -e qtivcomposer name=mix
sink_0::position="<0, 0>" sink_0::dimensions="<240, 270>" \
sink_1::position="<240, 0>" sink_1::dimensions="<240, 270>" \
sink_2::position="<480, 0>" sink_2::dimensions="<240, 270>" \
sink_3::position="<720, 0>" sink_3::dimensions="<240, 270>" \
sink_4::position="<960, 0>" sink_4::dimensions="<240, 270>" \
sink_5::position="<1200, 0>" sink_5::dimensions="<240, 270>" \
sink_6::position="<1440, 0>" sink_6::dimensions="<240, 270>" \
sink_7::position="<1680, 0>" sink_7::dimensions="<240, 270>" \
sink_8::position="<0, 270>" sink_8::dimensions="<240, 270>" \
sink_9::position="<240, 270>" sink_9::dimensions="<240, 270>" \
sink_10::position="<480, 270>" sink_10::dimensions="<240, 270>" \
sink_11::position="<720, 270>" sink_11::dimensions="<240, 270>" \
sink_12::position="<960, 270>" sink_12::dimensions="<240, 270>" \
sink_13::position="<1200, 270>" sink_13::dimensions="<240, 270>" \
sink_14::position="<1440, 270>" sink_14::dimensions="<240, 270>" \
sink_15::position="<1680, 270>" sink_15::dimensions="<240, 270>" \
sink_16::position="<0, 540>" sink_16::dimensions="<240, 270>" \
sink_17::position="<240, 540>" sink_17::dimensions="<240, 270>" \
sink_18::position="<480, 540>" sink_18::dimensions="<240, 270>" \
sink_19::position="<720, 540>" sink_19::dimensions="<240, 270>" \
sink_20::position="<960, 540>" sink_20::dimensions="<240, 270>" \
sink_21::position="<1200, 540>" sink_21::dimensions="<240, 270>" \
sink_22::position="<1440, 540>" sink_22::dimensions="<240, 270>" \
sink_23::position="<1680, 540>" sink_23::dimensions="<240, 270>" \
mix. ! fpsdisplaysink text-overlay=false video-sink="waylandsink
fullscreen=true" -v \
filesrc location=/opt/sjxbqm_1080p30fps_25_4882Kbps.mp4 ! qtdemux !
av1parse ! v4l2av1dec capture-io-mode=4 output-io-mode=4 ! mix. \
```

```
filesrc location=/opt/sjxbqm_1080p30fps_25_4882Kbps.mp4 ! qtdemux !
av1parse ! v4l2av1dec capture-io-mode=4 output-io-mode=4 ! mix. \
filesrc location=/opt/sjxbqm_1080p30fps_25_4882Kbps.mp4 ! qtdemux !
av1parse ! v4l2av1dec capture-io-mode=4 output-io-mode=4 ! mix. \
filesrc location=/opt/sjxbqm_1080p30fps_25_4882Kbps.mp4 ! qtdemux !
av1parse ! v4l2av1dec capture-io-mode=4 output-io-mode=4 ! mix. \
filesrc location=/opt/sjxbqm_1080p30fps_25_4882Kbps.mp4 ! qtdemux !
av1parse ! v4l2av1dec capture-io-mode=4 output-io-mode=4 ! mix. \
filesrc location=/opt/sjxbqm_1080p30fps_25_4882Kbps.mp4 ! qtdemux !
av1parse ! v4l2av1dec capture-io-mode=4 output-io-mode=4 ! mix. \
filesrc location=/opt/sjxbqm_1080p30fps_25_4882Kbps.mp4 ! qtdemux !
av1parse ! v4l2av1dec capture-io-mode=4 output-io-mode=4 ! mix. \
filesrc location=/opt/sjxbqm_1080p30fps_25_4882Kbps.mp4 ! qtdemux !
av1parse ! v4l2av1dec capture-io-mode=4 output-io-mode=4 ! mix. \
filesrc location=/opt/sjxbqm_1080p30fps_25_4882Kbps.mp4 ! qtdemux !
av1parse ! v4l2av1dec capture-io-mode=4 output-io-mode=4 ! mix. \
filesrc location=/opt/sjxbqm_1080p30fps_25_4882Kbps.mp4 ! qtdemux !
av1parse ! v4l2av1dec capture-io-mode=4 output-io-mode=4 ! mix. \
filesrc location=/opt/sjxbqm_1080p30fps_25_4882Kbps.mp4 ! qtdemux !
av1parse ! v4l2av1dec capture-io-mode=4 output-io-mode=4 ! mix. \
filesrc location=/opt/sjxbqm_1080p30fps_25_4882Kbps.mp4 ! qtdemux !
av1parse ! v4l2av1dec capture-io-mode=4 output-io-mode=4 ! mix. \
filesrc location=/opt/sjxbqm_1080p30fps_25_4882Kbps.mp4 ! qtdemux !
av1parse ! v4l2av1dec capture-io-mode=4 output-io-mode=4 ! mix. \
filesrc location=/opt/sjxbqm_1080p30fps_25_4882Kbps.mp4 ! qtdemux !
av1parse ! v4l2av1dec capture-io-mode=4 output-io-mode=4 ! mix. \
filesrc location=/opt/sjxbqm_1080p30fps_25_4882Kbps.mp4 ! qtdemux !
av1parse ! v4l2av1dec capture-io-mode=4 output-io-mode=4 ! mix. \
filesrc location=/opt/sjxbqm_1080p30fps_25_4882Kbps.mp4 ! qtdemux !
av1parse ! v4l2av1dec capture-io-mode=4 output-io-mode=4 ! mix. \
filesrc location=/opt/sjxbqm_1080p30fps_25_4882Kbps.mp4 ! qtdemux !
av1parse ! v4l2av1dec capture-io-mode=4 output-io-mode=4 ! mix. \
filesrc location=/opt/sjxbqm_1080p30fps_25_4882Kbps.mp4 ! qtdemux !
av1parse ! v4l2av1dec capture-io-mode=4 output-io-mode=4 ! mix. \
filesrc location=/opt/sjxbqm_1080p30fps_25_4882Kbps.mp4 ! qtdemux !
av1parse ! v4l2av1dec capture-io-mode=4 output-io-mode=4 ! mix. \
filesrc location=/opt/sjxbqm_1080p30fps_25_4882Kbps.mp4 ! qtdemux !
av1parse ! v4l2av1dec capture-io-mode=4 output-io-mode=4 ! mix. \
filesrc location=/opt/sjxbqm_1080p30fps_25_4882Kbps.mp4 ! qtdemux !
av1parse ! v4l2av1dec capture-io-mode=4 output-io-mode=4 ! mix. \
filesrc location=/opt/sjxbqm_1080p30fps_25_4882Kbps.mp4 ! qtdemux !
av1parse ! v4l2av1dec capture-io-mode=4 output-io-mode=4 ! mix. \
filesrc location=/opt/sjxbqm_1080p30fps_25_4882Kbps.mp4 ! qtdemux !
av1parse ! v4l2av1dec capture-io-mode=4 output-io-mode=4 ! mix. \
filesrc location=/opt/sjxbqm_1080p30fps_25_4882Kbps.mp4 ! qtdemux !
av1parse ! v4l2av1dec capture-io-mode=4 output-io-mode=4 ! mix. \
filesrc location=/opt/sjxbqm_1080p30fps_25_4882Kbps.mp4 ! qtdemux !
av1parse ! v4l2av1dec capture-io-mode=4 output-io-mode=4 ! mix. \
filesrc location=/opt/sjxbqm_1080p30fps_25_4882Kbps.mp4 ! qtdemux !
av1parse ! v4l2av1dec capture-io-mode=4 output-io-mode=4 ! mix.
```

H.264–12 sessions decode and encode

```
ulimit -n 4096 && gst-launch-1.0 -e filesrc location=/opt/
1920_1080_h264_30fps_1min.mp4 ! qtdemux ! h264parse ! v4l2h264dec capture-io-
mode=4 output-io-mode=4 ! v4l2h264enc capture-io-mode=4 output-io-mode=5 !
h264parse ! mp4mux ! filesink location=/opt/enc_1.mp4 \
filesrc location=/opt/1920_1080_h264_30fps_1min.mp4 ! qtdemux ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! v4l2h264enc capture-io-
mode=4 output-io-mode=5 ! h264parse ! mp4mux ! filesink location=/opt/
enc_2.mp4 \
filesrc location=/opt/1920_1080_h264_30fps_1min.mp4 ! qtdemux ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! v4l2h264enc capture-io-
mode=4 output-io-mode=5 ! h264parse ! mp4mux ! filesink location=/opt/
enc_3.mp4 \
filesrc location=/opt/1920_1080_h264_30fps_1min.mp4 ! qtdemux ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! v4l2h264enc capture-io-
mode=4 output-io-mode=5 ! h264parse ! mp4mux ! filesink location=/opt/
enc_4.mp4 \
filesrc location=/opt/1920_1080_h264_30fps_1min.mp4 ! qtdemux ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! v4l2h264enc capture-io-
mode=4 output-io-mode=5 ! h264parse ! mp4mux ! filesink location=/opt/
enc_5.mp4 \
filesrc location=/opt/1920_1080_h264_30fps_1min.mp4 ! qtdemux ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! v4l2h264enc capture-io-
mode=4 output-io-mode=5 ! h264parse ! mp4mux ! filesink location=/opt/
enc_6.mp4 \
filesrc location=/opt/1920_1080_h264_30fps_1min.mp4 ! qtdemux ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! v4l2h264enc capture-io-
mode=4 output-io-mode=5 ! h264parse ! mp4mux ! filesink location=/opt/
enc_7.mp4 \
filesrc location=/opt/1920_1080_h264_30fps_1min.mp4 ! qtdemux ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! v4l2h264enc capture-io-
mode=4 output-io-mode=5 ! h264parse ! mp4mux ! filesink location=/opt/
enc_8.mp4 \
filesrc location=/opt/1920_1080_h264_30fps_1min.mp4 ! qtdemux ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! v4l2h264enc capture-io-
mode=4 output-io-mode=5 ! h264parse ! mp4mux ! filesink location=/opt/
enc_9.mp4 \
filesrc location=/opt/1920_1080_h264_30fps_1min.mp4 ! qtdemux ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! v4l2h264enc capture-io-
mode=4 output-io-mode=5 ! h264parse ! mp4mux ! filesink location=/opt/
enc_10.mp4 \
filesrc location=/opt/1920_1080_h264_30fps_1min.mp4 ! qtdemux ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! v4l2h264enc capture-io-
mode=4 output-io-mode=5 ! h264parse ! mp4mux ! filesink location=/opt/
enc_11.mp4 \
filesrc location=/opt/1920_1080_h264_30fps_1min.mp4 ! qtdemux ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! v4l2h264enc capture-io-
```

```
mode=4 output-io-mode=5 ! h264parse ! mp4mux ! filesink location=/opt/
enc_12.mp4
```

H264–24 sessions encode

```
gst-launch-1.0 -e filesrc blocksize=3133440 location=/tmp/dump_1080.yuv !
video/x-raw,format="NV12",width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4 output-io-
mode=5 ! filesink location=/opt/enc_0.h264 \
filesrc blocksize=3133440 location=/tmp/dump_1080.yuv ! video/x-
raw,format="NV12",width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4 output-io-
mode=5 ! filesink location=/opt/enc_1.h264 \
filesrc blocksize=3133440 location=/tmp/dump_1080.yuv ! video/x-
raw,format="NV12",width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4 output-io-
mode=5 ! filesink location=/opt/enc_2.h264 \
filesrc blocksize=3133440 location=/tmp/dump_1080.yuv ! video/x-
raw,format="NV12",width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4 output-io-
mode=5 ! filesink location=/opt/enc_3.h264 \
filesrc blocksize=3133440 location=/tmp/dump_1080.yuv ! video/x-
raw,format="NV12",width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4 output-io-
mode=5 ! filesink location=/opt/enc_4.h264 \
filesrc blocksize=3133440 location=/tmp/dump_1080.yuv ! video/x-
raw,format="NV12",width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4 output-io-
mode=5 ! filesink location=/opt/enc_5.h264 \
filesrc blocksize=3133440 location=/tmp/dump_1080.yuv ! video/x-
raw,format="NV12",width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4 output-io-
mode=5 ! filesink location=/opt/enc_6.h264 \
filesrc blocksize=3133440 location=/tmp/dump_1080.yuv ! video/x-
raw,format="NV12",width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4 output-io-
mode=5 ! filesink location=/opt/enc_7.h264 \
filesrc blocksize=3133440 location=/tmp/dump_1080.yuv ! video/x-
raw,format="NV12",width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4 output-io-
mode=5 ! filesink location=/opt/enc_8.h264 \
filesrc blocksize=3133440 location=/tmp/dump_1080.yuv ! video/x-
raw,format="NV12",width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4 output-io-
mode=5 ! filesink location=/opt/enc_9.h264 \
filesrc blocksize=3133440 location=/tmp/dump_1080.yuv ! video/x-
raw,format="NV12",width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4 output-io-
```

```
mode=5 ! filesink location=/opt/enc_10.h264 \
filesrc blocksize=3133440 location=/tmp/dump_1080.yuv ! video/x-
raw,format="NV12",width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4 output-io-
mode=5 ! filesink location=/opt/enc_11.h264 \
filesrc blocksize=3133440 location=/tmp/dump_1080.yuv ! video/x-
raw,format="NV12",width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4 output-io-
mode=5 ! filesink location=/opt/enc_12.h264 \
filesrc blocksize=3133440 location=/tmp/dump_1080.yuv ! video/x-
raw,format="NV12",width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4 output-io-
mode=5 ! filesink location=/opt/enc_13.h264 \
filesrc blocksize=3133440 location=/tmp/dump_1080.yuv ! video/x-
raw,format="NV12",width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4 output-io-
mode=5 ! filesink location=/opt/enc_14.h264 \
filesrc blocksize=3133440 location=/tmp/dump_1080.yuv ! video/x-
raw,format="NV12",width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4 output-io-
mode=5 ! filesink location=/opt/enc_15.h264 \
filesrc blocksize=3133440 location=/tmp/dump_1080.yuv ! video/x-
raw,format="NV12",width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4 output-io-
mode=5 ! filesink location=/opt/enc_16.h264 \
filesrc blocksize=3133440 location=/tmp/dump_1080.yuv ! video/x-
raw,format="NV12",width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4 output-io-
mode=5 ! filesink location=/opt/enc_17.h264 \
filesrc blocksize=3133440 location=/tmp/dump_1080.yuv ! video/x-
raw,format="NV12",width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4 output-io-
mode=5 ! filesink location=/opt/enc_18.h264 \
filesrc blocksize=3133440 location=/tmp/dump_1080.yuv ! video/x-
raw,format="NV12",width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4 output-io-
mode=5 ! filesink location=/opt/enc_19.h264 \
filesrc blocksize=3133440 location=/tmp/dump_1080.yuv ! video/x-
raw,format="NV12",width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4 output-io-
mode=5 ! filesink location=/opt/enc_20.h264 \
filesrc blocksize=3133440 location=/tmp/dump_1080.yuv ! video/x-
raw,format="NV12",width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4 output-io-
mode=5 ! filesink location=/opt/enc_21.h264 \
filesrc blocksize=3133440 location=/tmp/dump_1080.yuv ! video/x-
raw,format="NV12",width=1920,height=1080,framerate=30/1,interlace-
```

```
mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4 output-io-
mode=5 ! filesink location=/opt/enc_22.h264 \
filesrc blocksize=3133440 location=/tmp/dump_1080.yuv ! video/x-
raw,format="NV12",width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4 output-io-
mode=5 ! filesink location=/opt/enc_23.h264
```

5.2.2.13 AV1 decoder and 4K VP9 decoder with 240 fps playback

Run the decoding use cases with the AV1 and VP9 decoders. These use cases are only applicable to QCS9075.

AV1 use cases

- 3840 × 2160

  ```
  gst-launch-1.0 -e filesrc location=/opt/
  SeaOtters_AV1_3840x2160_30_25.0Mbps_Main_no_audio_30.0.mp4 ! qtdemux !
  av1parse ! v4l2av1dec capture-io-mode=4 output-io-mode=4 ! fpsdisplaysink
  video-sink="waylandsink fullscreen=true" -v
  ```

- 1920 × 1080

  ```
  gst-launch-1.0 -e filesrc location=/opt/
  CaribouHerd_AV1_1920x1080_30_39.38Mbps_Main_AAC_34.472.mp4 ! qtdemux !
  av1parse ! v4l2av1dec capture-io-mode=4 output-io-mode=4 ! fpsdisplaysink
  video-sink="waylandsink fullscreen=true" -v
  ```

- 1280 × 720

  ```
  gst-launch-1.0 -e filesrc location=/opt/av1.mp4 ! qtdemux ! av1parse !
  v4l2av1dec capture-io-mode=4 output-io-mode=4 ! fpsdisplaysink video-
  sink="waylandsink fullscreen=true" -v
  ```

VP9 use cases

3840 × 2160 with 240 fps

```
gst-launch-1.0 -e filesrc location=/opt/test_UHD_240FPS.ivf ! ivfparse !
queue ! v4l2vp9dec capture-io-mode=4 output-io-mode=4 ! video/x-
raw,format=NV12 ! queue ! fpsdisplaysink text-overlay=false video-
sink="fakesink" -v
```

5.2.3 Multi-camera/Multi-client use cases

The multi-camera/multi-client use cases show scenarios where several camera streams are displayed in modes such as picture-in-picture and side by side.

Before you run the use cases, ensure that you complete the preconditions mentioned in Run multimedia use cases.

GMSL camera limitation for QCS8275 and QCS9075

With the existing software configuration, a single GMSL port supports connection to only one GMSL camera on any of its four slots.

- The GMSL Port-0 supports a single Bayer camera.
- The current release supports the OX03f10 Bayer GMSL and OX03f10 YUV GMSL cameras.

Table 5-27 QCS8275 and QCS9075–GSML port support and limitation

SoC	GMSL port connection
QCS6490	Ensure that you connect the MIPI cameras to the CSI slots on the RB3 Gen 2 device. ▪ Connect the OV9282 MIPI camera to the CAM 0A slot. ▪ Connect the IMX577 MPI camera to the CAM3 slot.
QCS8275	Ensure that you connect the GMSL camera sensors to the Ride SX device. ▪ Connect the OX03F10 Bayer GMSL camera to GMSL port 0. ▪ Connect the OX03F10 YUV GMSL cameras to port 1 and port 2.
QCS9075	▪ MIPI: Connect four OV9282 MIPI cameras to the CSI slots. ▪ GMSL: Connect the OX03F10 Bayer GMSL camera to GMSL port 0 and OX03F10 YUV GMSL cameras to port 2 and port 3.

5.2.3.1 Two streams—4k AVC MP4, 1080p YUV preview as Client 1 (main camera) and 720p AVC, 720p YUV as Client 2 (secondary camera)

The use cases use two (4k and 1080p) streams from the main camera, and two 720p streams from the secondary camera. One stream from each camera is multiplexed and encoded while the other pair is displayed.

Table 5-28 Multi camera client use cases for encoding and preview

Use case	Camera (client)
Multiplexed and encoded	▪ Main camera–4k AVC MP4 ▪ Secondary camera–720p AVC
Preview	▪ Main camera–1080p YUV ▪ Secondary camera–720p YUV

Run the pipeline:

```
gst-launch-1.0 -e qtiqmmfsrc name=camsrc_0 video_1::type=preview ! video/x-
raw,format=NV12,width=3840,height=2160,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! queue ! v4l2h264enc capture-io-mode=4
output-io-mode=5  ! queue ! h264parse ! mp4mux ! queue ! filesink
location="/opt/mux1.mp4" camsrc_0. ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1 ! waylandsink
sync=false qtiqmmfsrc name=camsrc_1 camera=1 video_1::type=preview ! video/x-
raw,format=NV12,width=1280,height=720,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! queue ! v4l2h264enc capture-io-mode=4
output-io-mode=5 ! queue ! h264parse ! mp4mux ! queue ! filesink
```

```
location="/opt/mux2.mp4" camsrc_1. ! video/x-raw, format=NV12, width=1280,
height=720, framerate=30/1 ! waylandsink sync=false
```

To stop the use case, use **CTRL + C**.

Pull recorded content from the device using the following `scp` commands and play content on the host computer.

```
scp -r root@[DEVICE IP-ADDR]:/opt/mux1.mp4 user1@[HOST IP-ADDR]:/workspace
```

```
scp -r root@[DEVICE IP-ADDR]:/opt/mux2.mp4 user1@[HOST IP-ADDR]:/workspace
```

You can play the MP4 file and snapshots on a media player and view the preview stream on the screen.

The figure shows the flow of pipeline execution:

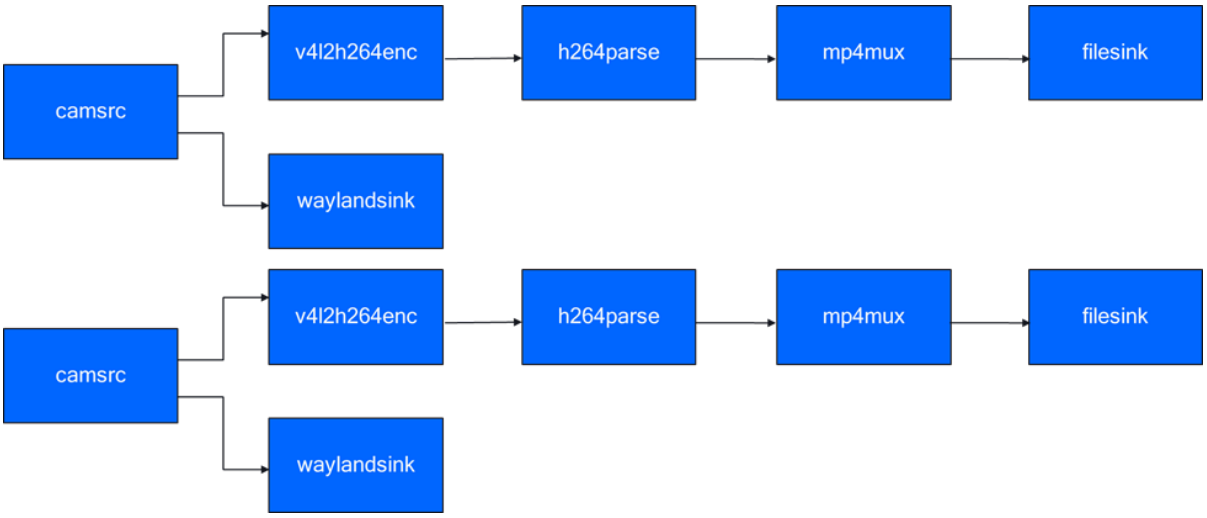


Figure 5-40 Pipeline for two camera clients–encoding and preview

5.2.3.2 Two 720p streams—one from each camera with side by side stitching for display

The pipeline shows two 720p streams from the main camera and secondary camera respectively. Both the streams are composed side by side and displayed.

Run the pipeline:

```
gst-launch-1.0 -e qtivcomposer name=mixer sink_0::position="<0, 0>"
sink_0::dimensions="<640, 360>" sink_1::position="<640, 0>"
sink_1::dimensions="<640, 360>" mixer. ! queue ! waylandsink enable-last-
sample=false fullscreen=true qtiqmmfsrc name=camsrc_0 camera=0 ! video/x-raw,
format=NV12, width=1280, height=720, framerate=30/1 ! mixer. qtiqmmfsrc
name=camsrc_1 camera=1 ! video/x-raw, format=NV12, width=1280, height=720,
framerate=30/1 ! mixer.
```

The composed streams from the camera are displayed.

To stop the use case, use **CTRL + C**.

The figure shows the flow of pipeline execution:

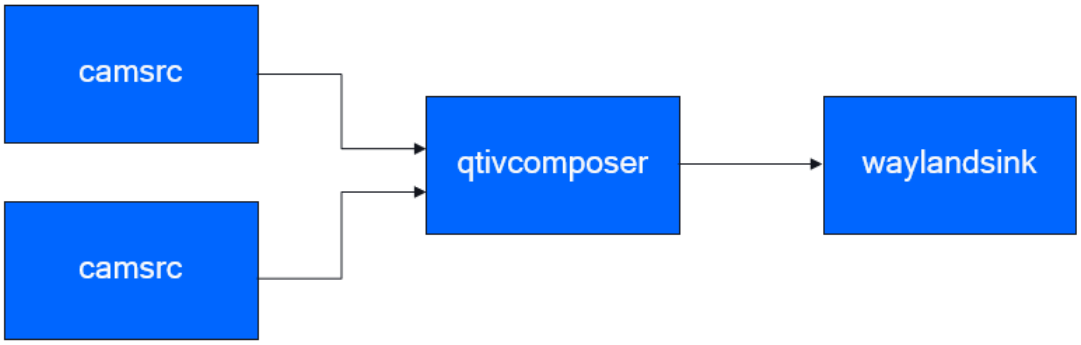


Figure 5-41 Two streams from primary and secondary cameras–side by side display

5.2.3.3 Two 720p streams—one from each camera with side by side stitching for encode and RTSP streaming

The pipeline shows two 720p streams from the main camera and secondary camera respectively. Both are composed side by side, and then duplicated and encoded. One is multiplexed and saved to a file while the other is streamed through RTSP.

Run the RTSP server in a separate console on target with udpsrc. You can also run it in the background as a service:

```
gst-rtsp-server -p 8900 -a <IP address of device> -m /live "( udpsrc
name=pay0 port=8554 caps=\"application/x-rtp,media=video,clock-
rate=90000,encoding-name=H264,payload=96\" )"
```

Run the pipeline in the same or another console on target:

```
gst-launch-1.0 -e qtivcomposer name=mixer sink_0::position="<0, 0>"
sink_0::dimensions="<640, 360>" sink_1::position="<640, 0>"
sink_1::dimensions="<640, 360>" mixer. ! queue ! tee name=t_split t_split. !
queue ! video/x-raw,interlace-mode=progressive,colorimetry=bt601 !
v4l2h264enc capture-io-mode=4 output-io-mode=5 ! h264parse ! queue ! mp4mux !
queue ! filesink location=/opt/mux.mp4 t_split. ! queue ! video/x-
raw,interlace-mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-
mode=4 output-io-mode=5 ! h264parse config-interval=-1 ! rtph264pay pt=96 !
udpsink host=127.0.0.1 port=8554 qtiqmmfsrc name=camsrc_0 camera=0 ! video/x-
raw, format=NV12, width=1280, height=720, framerate=30/1 ! mixer. qtiqmmfsrc
name=camsrc_1 camera=1 ! video/x-raw, format=NV12, width=1280, height=720,
framerate=30/1 ! mixer.
```

To stop the use case, use **CTRL + C**.

Pull the recorded content from the device using the `scp` command, and play content on host computer.

```
scp root@<IP address of target device>:/opt/mux.mp4 <destination directory>
```

You can play the MP4 file on a media player. The streamed video is displayed as an RTSP stream on the host computer. See View RTSP stream on host computer.

The figure shows the flow of the pipeline execution:

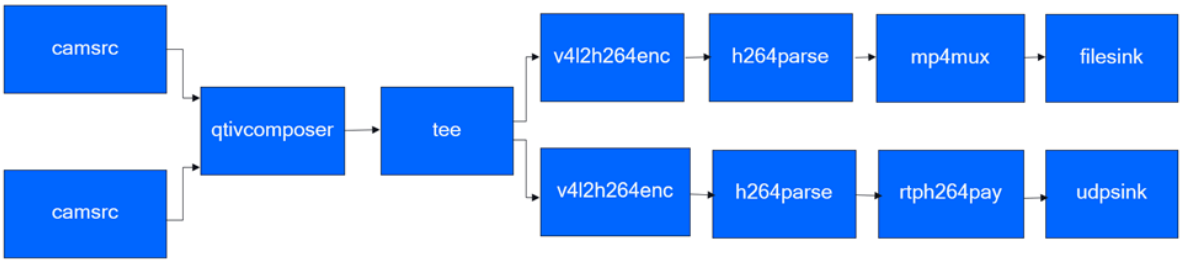


Figure 5-42 Pipeline for two 720p camera streams–side by side display, encoded, and streamed through RTSP

5.2.3.4 Two 720p streams—one from each camera with picture-in-picture composition and sent to display

The use case uses two 720p streams from the main camera and secondary camera respectively. Both are composed as picture-in-picture and displayed.

Run the pipeline:

```
gst-launch-1.0 -e qtivcomposer name=mixer sink_0::position="<0, 0>"
sink_0::dimensions="<1280, 720>" sink_1::position="<590, 310>"
sink_1::dimensions="<640, 360>" mixer. ! queue ! waylandsink enable-last-
sample=false fullscreen=true qtiqmmfsrc name=camsrc_0 camera=0 ! video/x-raw,
format=NV12, width=1280, height=720, framerate=30/1 ! mixer. qtiqmmfsrc
name=camsrc_1 camera=1 ! video/x-raw, format=NV12, width=1280, height=720,
framerate=30/1 ! mixer.
```

The composed streams from the camera (picture-in-picture) are displayed.

To stop the use case, use **CTRL + C**.

The figure shows the flow of the pipeline execution:

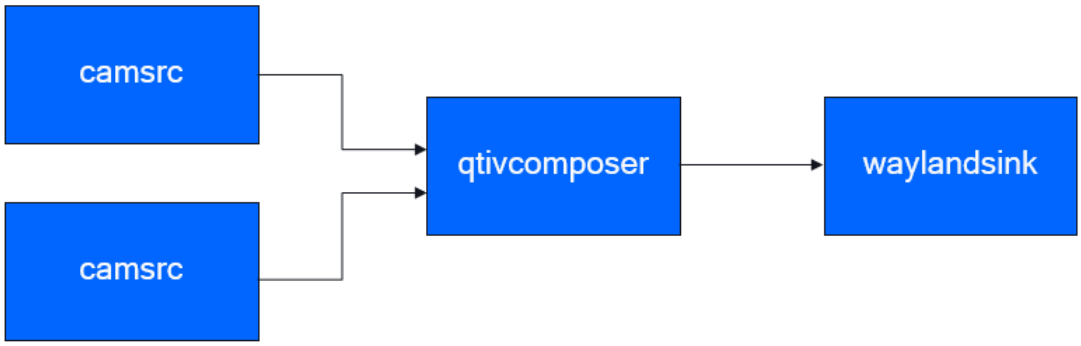


Figure 5-43 Pipeline for two 720p streams–picture-in-picture display

5.2.3.5 Two 720p streams—one from each camera with picture-in-picture composition for encode and RTSP streaming

The use case uses two 720p streams from the main camera and secondary camera respectively. Both are composed as picture-in-picture, and then duplicated and encoded. One stream is multiplexed and saved to a file and the other is streamed through RTSP.

Run the pipeline:

- Run RTSP server in a separate console on target with udpsrc. You can run it in the background as a service:

```
gst-rtsp-server -p 8900 -a <IP address of device> -m /live "( udpsrc
name=pay0 port=8554 caps=\"application/x-rtp,media=video,clock-
rate=90000,encoding-name=H264,payload=96\" )"
```

- Run the pipeline in same or another console on target:

```
gst-launch-1.0 -e qtivcomposer name=mixer sink_0::position="<0, 0>"
sink_0::dimensions="<1280, 720>" sink_1::position="<590, 310>"
sink_1::dimensions="<640, 360>" mixer. ! queue ! tee name=t_split
t_split. ! queue ! video/x-raw,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4 output-
io-mode=5 ! h264parse ! queue ! mp4mux ! queue ! filesink location=/opt/
mux.mp4 t_split. ! queue ! video/x-raw,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4 output-
io-mode=5 ! h264parse config-interval=-1 ! rtph264pay pt=96 ! udpsink
host=<ip_address> port=8554 qtiqmmfsrc name=camsrc_0 camera=0 ! video/x-
raw, format=NV12, width=1280, height=720, framerate=30/1 ! mixer.
qtiqmmfsrc name=camsrc_1 camera=1 ! video/x-raw, format=NV12, width=1280,
height=720, framerate=30/1 ! mixer.
```

- To stop the use case, use **CTRL + C**.

  Pull the recorded content from the device using the `scp` command, and play content on host computer.

```
scp root@<IP address of target device>:/opt/ <destination directory>
```

You can play the MP4 file on a media player. The video is streamed over RTSP on a host computer. See View RTSP stream on host computer.

The figure shows the flow of the pipeline execution:

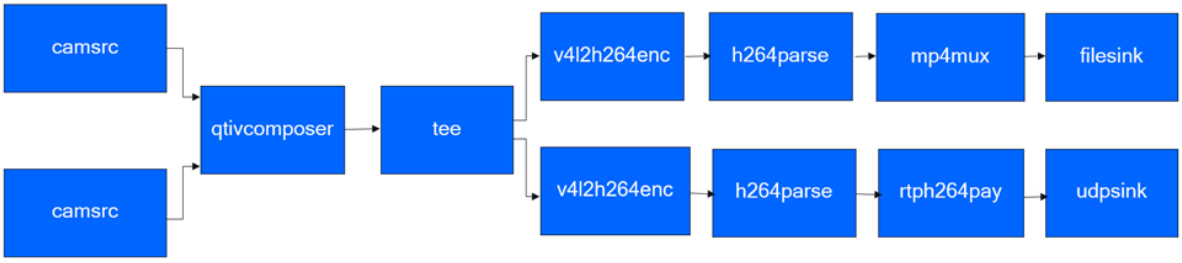


Figure 5-44 Pipeline for two 720p camera streams–picture-in-picture display, encoded, and streamed through RTSP

RELATED INFORMATION

"Multi-camera streaming" on page 115
"Multi-camera streaming using Python" on page 181

5.2.4 Transform and Transcode use cases

The transcoding use cases show converting or transforming the video streams from one format to another.

Before you run the use cases, ensure that you complete the preconditions mentioned in Run multimedia use cases.

NOTE The Qualcomm Computer Vision SDK (fcv) engine is currently not supported.

5.2.4.1 1080p to 1080p–rotate (90/180/270)

The use case shows the rotation of scenes from a camera by 180 degrees. The rotated image is displayed on a local display device.

NOTE To rotate the video at different angles, use the rotate property from the qtivtransform plugin.

Use the following commands to run the use case:

- Transform the scene running on DSP using Qualcomm Computer Vision SDK (fcv) engine:

```
gst-launch-1.0 -e qtiqmmfsrc name=camsrc video_0::type=video
video_1::type=preview ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1 ! qtivtransform
engine=fcv rotate=180 ! waylandsink fullscreen=true async=false sync=false
```

- Transform the scene running on the GPU using OpenGL ES:

```
gst-launch-1.0 -e qtiqmmfsrc name=camsrc video_0::type=video
video_1::type=preview ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1 ! qtivtransform
engine=gles rotate=180 ! waylandsink fullscreen=true async=false sync=false
```

The host camera displays the transformed stream.

To stop the use case, use **CTRL + C**.

The figure shows the flow of the pipeline execution:

Figure 5-45 Pipeline for rotation and display–camera source

5.2.4.2 Offline 1080p stream to 1080p–rotate (90/180/270)

The use case shows the rotation of scenes from a file source by 180 degrees. The rotated scene is displayed on a local display device.

NOTE To rotate the video at different angles, see the rotate property from qtivtransform.

Use the following commands to run the use case:

- Transform the scene running on DSP using Qualcomm Computer Vision SDK (fcv) engine:

```
gst-launch-1.0 -e filesrc location=/opt/<H264/AVC_file>.mp4 ! qtdemux !
queue ! h264parse ! v4l2h264dec capture-io-mode=4 output-io-mode=4 !
video/x-raw,format=NV12 ! qtivtransform engine=fcv rotate=180 ! queue !
waylandsink fullscreen=true
```

- Transform the scene running on the GPU using OpenGL ES:

```
gst-launch-1.0 -e filesrc location=/opt/<H264/AVC_file>.mp4 ! qtdemux !
queue ! h264parse ! v4l2h264dec capture-io-mode=4 output-io-mode=4 !
video/x-raw,format=NV12 ! qtivtransform engine=gles rotate=180 ! queue !
waylandsink fullscreen=true
```

The transformed stream from the file source is displayed.

To stop the use case, use **CTRL + C**.

The figure shows the flow of the pipeline execution:

Figure 5-46 Pipeline for rotation and display–file source

5.2.4.3 4K to 1080p–rotate (90/180/270) and downscale

The use case shows a 4k resolution video stream from a camera downscaled to 1080p and rotated by 180 degrees. The video is then displayed on a local display device.

NOTE To rotate the video at different angles, see the rotate property from qtivtransform.

Use the following commands to run the use case:

- Qualcomm Computer Vision SDK (fcv) engine:

```
gst-launch-1.0 -e qtiqmmfsrc name=camsrc video_0::type=video
video_1::type=preview ! video/x-
raw,format=NV12,width=3840,height=2160,framerate=30/1 ! qtivtransform
engine=fcv rotate=180 ! video/x-raw,format=NV12,width=1920,height=1080 !
waylandsink fullscreen=true
```

- GLES engine:

```
gst-launch-1.0 -e qtiqmmfsrc name=camsrc video_0::type=video
video_1::type=preview ! video/x-
raw,format=NV12,width=3840,height=2160,framerate=30/1 ! qtivtransform
engine=gles rotate=180 ! video/x-raw,format=NV12,width=1920,height=1080 !
waylandsink fullscreen=true
```

The transformed stream from the camera is displayed.

To stop the use case, use **CTRL + C**.

The figure shows the flow of the pipeline execution:

Figure 5-47 Pipeline for 4k image downscaled to 1080p, rotated, and displayed–camera source

5.2.4.4 Offline 4K to 1080p–rotate (90/180/270) and downscale

The pipeline shows a 4k resolution video stream from a file source downscaled to 1080p and rotated by 180 degrees. The video is then displayed on a local display device.

NOTE To rotate the video at different angles, see the rotate property from qtivtransform.

Use the following commands to run the use case:

- Qualcomm Computer Vision SDK (fcv) engine:

```
gst-launch-1.0 -e qtiqmmfsrc name=camsrc video_0::type=video
video_1::type=preview ! video/x-
raw,format=NV12,width=3840,height=2160,framerate=30/1 ! qtivtransform
engine=fcv rotate=180 ! video/x-raw,format=NV12,width=1920,height=1080 !
waylandsink fullscreen=true
```

- GLES engine:

```
gst-launch-1.0 -e filesrc location=/opt/<4K_H264/AVC_file>.mp4 ! qtdemux !
queue ! h264parse ! v4l2h264dec capture-io-mode=4 output-io-mode=4 !
```

```
video/x-raw,format=NV12 ! qtivtransform engine=gles rotate=180 ! video/x-
raw,format=NV12,width=1920,height=1080 ! waylandsink fullscreen=true
```

The transformed stream from the file source is displayed.

To stop the use case, use **CTRL + C**.

The figure shows the flow of the pipeline execution:

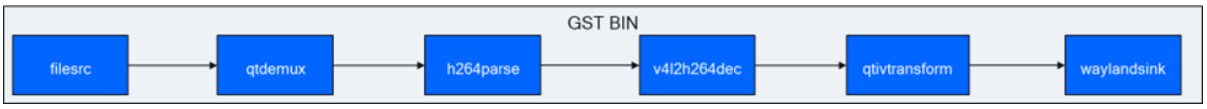


Figure 5-48 Pipeline for 4k stream downscaled to 1080p, and, rotated, and displayed–file source

5.2.4.5 1080p to 1080p–horizontal and vertical flip

The use case shows how to flip the scenes of a camera stream horizontally.

NOTE To flip the scenes, in the following commands, replace `flip-horizontal` with `flip-vertical` and vice versa. For more information, see qtivtransform.

Use the following commands to run the use case:

- Qualcomm Computer Vision SDK (fcv) engine:

```
gst-launch-1.0 -e qtiqmmfsrc name=camsrc video_0::type=video
video_1::type=preview ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1 ! qtivtransform
engine=fcv flip-horizontal=true ! waylandsink fullscreen=true
```

- GLES engine:

```
gst-launch-1.0 -e qtiqmmfsrc name=camsrc video_0::type=video
video_1::type=preview ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1 ! qtivtransform
engine=gles flip-horizontal=true ! waylandsink fullscreen=true
```

The transformed stream from a camera is displayed.

To stop the use case, use **CTRL + C**.

The figure shows the flow of the pipeline execution:

Figure 5-49 Pipeline for displaying flipped images–camera source

5.2.4.6 Offline 1080p to 1080p–horizontal and vertical flip

The use case shows how to flip scenes horizontally.

NOTE To flip the scenes, in the following commands, replace `flip-horizontal` with `flip-vertical` and vice versa. For more information, see qtivtransform.

Use the following commands to run the use case:

- Qualcomm Computer Vision SDK (fcv) engine:

```
gst-launch-1.0 -e filesrc location=/opt/<1080p_H264/AVC_file>.mp4 !
qtdemux ! queue ! h264parse ! v4l2h264dec capture-io-mode=4 output-io-
mode=4 ! video/x-raw,format=NV12 ! qtivtransform engine=fcv flip-
horizontal=true ! waylandsink fullscreen=true
```

- GLES engine:

```
gst-launch-1.0 -e filesrc location=/opt/<1080p_H264/AVC_file>.mp4 !
qtdemux ! queue ! h264parse ! v4l2h264dec capture-io-mode=4 output-io-
mode=4 ! video/x-raw,format=NV12 ! qtivtransform engine=gles flip-
horizontal=true ! waylandsink fullscreen=true
```

The transformed stream from a file source is displayed.

To stop the use case, use **CTRL + C**.

The figure shows the flow of the pipeline execution:

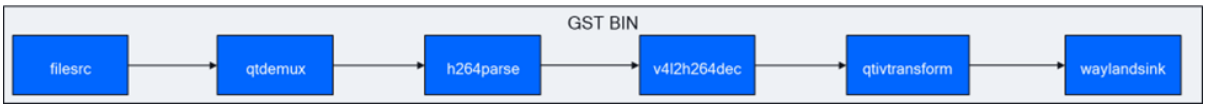


Figure 5-50 Pipeline for displaying flipped images–file source

5.2.4.7 4K to 1080p–horizontal and vertical flip with downscale

The use case shows how to downscale a 4k resolution video stream to 1080p and flip the scenes horizontally.

NOTE To flip the scenes, in the following commands, replace `flip-horizontal` with `flip-vertical` and vice versa. For more information, see qtivtransform.

Use the following commands to run the use case:

- Qualcomm Computer Vision SDK (fcv) engine:

```
gst-launch-1.0 -e qtiqmmfsrc name=camsrc video_0::type=video
video_1::type=preview ! video/x-
raw,format=NV12,width=3840,height=2160,framerate=30/1 ! qtivtransform
engine=fcv flip-vertical=true ! video/x-
raw,format=NV12,width=1920,height=1080 ! waylandsink fullscreen=true
```

- GLES engine:

```
gst-launch-1.0 -e qtiqmmfsrc name=camsrc video_0::type=video
video_1::type=preview ! video/x-
raw,format=NV12,width=3840,height=2160,framerate=30/1 ! qtivtransform
```

```
engine=gles flip-vertical=true ! video/x-
raw,format=NV12,width=1920,height=1080 ! waylandsink fullscreen=true
```

The transformed stream from the camera is displayed.

To stop the use case, use **CTRL + C**.

The figure shows the flow of the pipeline execution:

Figure 5-51 Pipeline for displaying flipped and downscaled images–camera source

5.2.4.8 Downscale 4K to 1080p and encode to AVC MP4

The use case uses two copies of a 4k resolution video stream. The first copy is encoded and multiplexed to an MP4 file. A second copy is shared with qtivtransform to downscale it to 1080p resolution. The 1080p stream is then encoded and multiplexed to an MP4 file on the device.

Use the following commands to run the use case:

- Qualcomm Computer Vision SDK (fcv) engine:

```
gst-launch-1.0 -e qtiqmmfsrc name=camsrc video_0::type=video
video_1::type=preview ! video/x-
raw,format=NV12,width=3840,height=2160,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! tee name=t_split t_split. ! queue !
v4l2h264enc capture-io-mode=4 output-io-mode=5 ! h264parse ! queue !
mp4mux ! queue ! filesink location=/opt/4k_video.mp4 t_split. !
qtivtransform engine=fcv ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! queue ! v4l2h264enc capture-io-mode=4
output-io-mode=5 ! h264parse ! queue ! mp4mux ! queue ! filesink
location=/opt/downscaled_1080p_video.mp4
```

- GLES engine:

```
gst-launch-1.0 -e qtiqmmfsrc name=camsrc video_0::type=video
video_1::type=preview ! video/x-
raw,format=NV12,width=3840,height=2160,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! tee name=t_split t_split. ! queue !
v4l2h264enc capture-io-mode=4 output-io-mode=5 ! h264parse ! queue !
mp4mux ! queue ! filesink location=/opt/4k_video.mp4 t_split. !
qtivtransform engine=gles ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! queue ! v4l2h264enc capture-io-mode=4
output-io-mode=5 ! h264parse ! queue ! mp4mux ! queue ! filesink
location=/opt/downscaled_1080p_video.mp4
```

To stop the use case, use **CTRL + C**.

Pull the recorded content from the device using the following `scp` command, and play content on the host computer.

```
scp root@<IP address of target device>:/opt/ <destination directory>
```

Pull the file content:

```
scp -r root@[DEVICE IP-ADDR]:/opt/4k_video.mp4 user1@[HOST IP-ADDR]:/workspace
```

```
scp -r root@[DEVICE IP-ADDR]:/opt/downscaled_1080p_video.mp4 user1@[HOST IP-ADDR]:/workspace
```

You can play the MP4 files on a media player. The camera stream is displayed.

The figure shows the flow of the pipeline execution:

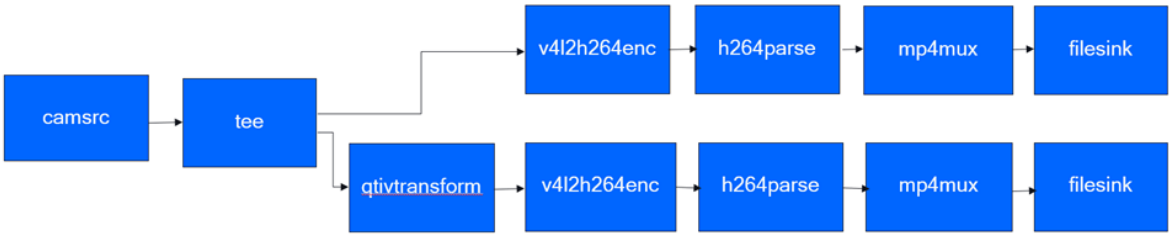


Figure 5-52 Pipeline for 4k video–encoding and downscaling to 1080p

5.2.4.9 More transform and transcode use cases

Video transform use cases

- 4K@30 AVC → decode → rotate and downscale → 1080p@30 AVC
  - Qualcomm Computer Vision SDK (fcv) engine:

    ```
    gst-launch-1.0 -e filesrc location=/opt/<4K_H264/AVC_file>.mp4 ! qtdemux ! queue ! h264parse ! v4l2h264dec capture-io-mode=5 output-io-mode=5 ! qtivtransform engine=fcv rotate=90CCW ! video/x-raw,format=NV12,width=1920,height=1080 ! v4l2h264enc capture-io-mode=4 output-io-mode=5 ! h264parse ! queue ! mp4mux ! queue ! filesink location=/opt/video.mp4
    ```

  - GLES engine:

    ```
    gst-launch-1.0 -e filesrc location=/opt/<4K_H264/AVC_file>.mp4 ! qtdemux ! queue ! h264parse ! v4l2h264dec capture-io-mode=5 output-io-mode=5 ! qtivtransform engine=gles rotate=90CCW ! video/x-raw,format=NV12,width=1920,height=1080 ! v4l2h264enc capture-io-mode=4 output-io-mode=5 ! h264parse ! queue ! mp4mux ! queue ! filesink location=/opt/video.mp4
    ```

- 4K@30 AVC → decode → flip and downscale → 1080p@30 AVC
  - Qualcomm Computer Vision SDK (fcv) engine:

    ```
    gst-launch-1.0 -e filesrc location=/opt/<4K_H264/AVC_file>.mp4 !
    qtdemux ! queue ! h264parse ! v4l2h264dec capture-io-mode=4 output-io-
    mode=4 ! qtivtransform engine=fcv flip-vertical=true ! video/x-
    raw,format=NV12,width=1920,height=1080 ! v4l2h264enc capture-io-mode=4
    output-io-mode=5 ! h264parse ! queue ! mp4mux ! queue ! filesink
    location=/opt/video.mp4
    ```

  - GLES engine:

    ```
    gst-launch-1.0 -e filesrc location=/opt/<4K_H264/AVC_file>.mp4 !
    qtdemux ! queue ! h264parse ! v4l2h264dec capture-io-mode=4 output-io-
    mode=4 ! qtivtransform engine=gles flip-vertical=true ! video/x-
    raw,format=NV12,width=1920,height=1080 ! v4l2h264enc capture-io-mode=4
    output-io-mode=5 ! h264parse ! queue ! mp4mux ! queue ! filesink
    location=/opt/video.mp4
    ```

- Video_qmmf_Transform_rotate_counter_clockwise_3840 x 2160 @30fps
  - Qualcomm Computer Vision SDK (fcv) engine:

    ```
    gst-launch-1.0 -e qtiqmmfsrc ! video/x-
    raw,format=NV12,width=3840,height=2160,framerate=30/1 ! qtivtransform
    engine=fcv rotate=90CCW ! queue ! v4l2h264enc capture-io-mode=4 output-
    io-mode=5 ! h264parse ! queue ! mp4mux ! queue ! filesink location=/opt/
    video.mp4
    ```

  - GLES engine:

    ```
    gst-launch-1.0 -e qtiqmmfsrc ! video/x-
    raw,format=NV12,width=3840,height=2160,framerate=30/1 ! qtivtransform
    engine=gles rotate=90CCW ! queue ! video/x-
    raw,format=NV12,width=3840,height=2160,framerate=30/1,interlace-
    mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4
    output-io-mode=5 ! h264parse ! queue ! mp4mux ! queue ! filesink
    location=/opt/video.mp4
    ```

- Video_qmmf_Transform_scale_down_3840 x 2160 @30fps_to_1080p
  - Qualcomm Computer Vision SDK (fcv) engine:

    ```
    gst-launch-1.0 -e qtiqmmfsrc ! video/x-
    raw,format=NV12,width=3840,height=2160,framerate=30/1 ! qtivtransform
    engine=fcv ! queue ! video/x-
    raw,format=NV12,width=1920,height=1080,framerate=30/1,interlace-
    mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4
    output-io-mode=5 ! h264parse ! queue ! mp4mux ! queue ! filesink
    location=/opt/video.mp4
    ```

  - GLES engine:

    ```
    gst-launch-1.0 -e qtiqmmfsrc ! video/x-
    raw,format=NV12,width=3840,height=2160,framerate=30/1 ! qtivtransform
    ```

```
engine=gles ! queue ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4
output-io-mode=5 ! h264parse ! queue ! mp4mux ! queue ! filesink
location=/opt/video.mp4
```

- Video_qmmf_Transform_flip_horizontal_3840 x 2160 @30fps
  - Qualcomm Computer Vision SDK (fcv) engine:

    ```
    gst-launch-1.0 -e qtiqmmfsrc ! video/x-
    raw,format=NV12,width=3840,height=2160,framerate=30/1 ! qtivtransform
    engine=fcv flip-horizontal=true ! video/x-
    raw,format=NV12,width=3840,height=2160,framerate=30/1,interlace-
    mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4
    output-io-mode=5 ! h264parse ! queue ! mp4mux ! queue ! filesink
    location=/opt/video_flip_h.mp4
    ```

  - GLES engine:

    ```
    gst-launch-1.0 -e qtiqmmfsrc ! video/x-
    raw,format=NV12,width=3840,height=2160,framerate=30/1 ! qtivtransform
    engine=gles flip-horizontal=true ! video/x-
    raw,format=NV12,width=3840,height=2160,framerate=30/1,interlace-
    mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4
    output-io-mode=5 ! h264parse ! queue ! mp4mux ! queue ! filesink
    location=/opt/video_flip_h.mp4
    ```

- Video_qmmf_Transform_flip_vertical_3840 x 2160 @30fps
  - Qualcomm Computer Vision SDK (fcv) engine:

    ```
    gst-launch-1.0 -e qtiqmmfsrc ! video/x-
    raw,format=NV12,width=3840,height=2160,framerate=30/1 ! qtivtransform
    engine=fcv flip-vertical=true ! video/x-
    raw,format=NV12,width=3840,height=2160,framerate=30/1,interlace-
    mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4
    output-io-mode=5 ! h264parse ! queue ! mp4mux ! queue ! filesink
    location=/opt/video_flip_v.mp4
    ```

  - GLES engine:

    ```
    gst-launch-1.0 -e qtiqmmfsrc ! video/x-
    raw,format=NV12,width=3840,height=2160,framerate=30/1 ! qtivtransform
    engine=gles flip-vertical=true ! video/x-
    raw,format=NV12,width=3840,height=2160,framerate=30/1,interlace-
    mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4
    output-io-mode=5 ! h264parse ! queue ! mp4mux ! queue ! filesink
    location=/opt/video_flip_v.mp4
    ```

Video transcode use cases

- 4k@30 HEVC → decode → 4k@30 AVC encode:

```
gst-launch-1.0 -e filesrc location=/opt/<4K_H265/HEVC_file>.mp4 !
qtdemux ! queue ! h265parse ! v4l2h265dec capture-io-mode=4 output-io-
mode=4 ! v4l2h264enc capture-io-mode=4 output-io-mode=5 ! queue !
h264parse ! mp4mux ! queue ! filesink location=/opt/offline_hevc_to_avc.mp4
```

- 4k@30 AVC → decode → 4k@30 HEVC encode:

```
gst-launch-1.0 -e filesrc location=/opt/<4K_H264/AVC_file>.mp4 ! qtdemux !
queue ! h264parse ! v4l2h264dec capture-io-mode=4 output-io-mode=4 !
v4l2h265enc capture-io-mode=4 output-io-mode=5 ! queue ! h265parse !
mp4mux ! queue ! filesink location="/opt/offline_avc_to_hevc.mp4"
```

Dynamic cropping using qtivtransform

Single 1080p file stream from which the ROI is cropped

1. Display the stream:

```
gst-launch-1.0 filesrc location=/opt/<input_file>.mp4 ! qtdemux ! queue !
h264parse ! v4l2h264dec capture-io-mode=4 output-io-mode=4 ! queue !
qtivtransform flip-vertical=true crop="<480,270,960,540>" ! waylandsink
fullscreen=true
```

2. Encode the stream data:

```
gst-launch-1.0 -e filesrc location=/opt/<input_file>.mp4 ! qtdemux !
queue ! h264parse ! v4l2h264dec capture-io-mode=4 output-io-mode=4 !
queue ! qtivtransform flip-horizontal=true crop="<480,270,960,540>" !
queue ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4 output-
io-mode=5 ! h264parse ! queue ! mp4mux ! queue ! filesink location=/opt/
offline_crop_vtransform.mp4
```

Single 1080p camera stream from which ROI is cropped

1. Display the stream:

```
gst-launch-1.0 -e qtiqmmfsrc ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1 ! queue !
qtivtransform flip-horizontal=true crop="<480,270,960,540>" ! waylandsink
fullscreen=true
```

2. Encode the stream data:

```
gst-launch-1.0 -e qtiqmmfsrc ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1 ! qtivtransform flip-
horizontal=true crop="<480,270,960,540>" ! queue ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4 output-
```

```
io-mode=5 ! h264parse ! queue ! mp4mux ! queue ! filesink location=/opt/
cam_crop_vtransform.mp4
```

RELATED INFORMATION

"Video transcoding" on page 155

5.2.5 Video playback use cases

The video playback use cases show the playback of the 4K video stream from a file, which is in a media container format such as MP4.

- Before you run the use cases, ensure that you complete the preconditions mentioned in Run multimedia use cases.
- Run all the commands in the SSH shell.
- To stop the use case, use **CTRL + C**.

NOTE To play the YUV file, pull the YUV dump from the device and play with a YUV player on the host computer. For example, you can select `YUV420planarNV12` format, and use `1920 & 1088` as stride and scanlines.

Single stream video playback (H.264)

- Decode and display a single stream (H.264):

```
gst-launch-1.0 -e filesrc location=/opt/<h264_file>.mp4 ! qtdemux !
queue ! h264parse ! v4l2h264dec capture-io-mode=4 output-io-mode=4 !
waylandsink fullscreen=true
```

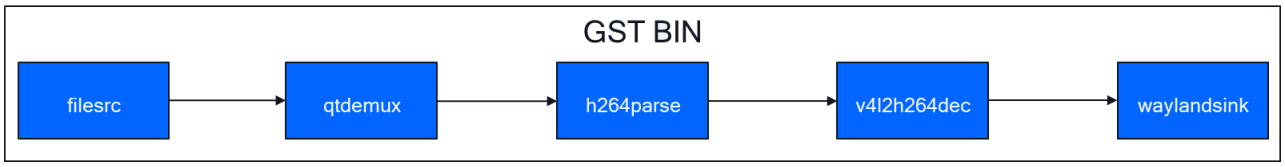


Figure 5-53 Pipeline to decode and display a single stream (H.264)

- Decode and save a stream to a file (H.264):

```
gst-launch-1.0 -e filesrc location=/opt/<h264_file>.mp4 ! qtdemux !
queue ! h264parse ! v4l2h264dec capture-io-mode=4 output-io-mode=4 !
video/x-raw, format=NV12 ! filesink location="/opt/video.yuv"
```

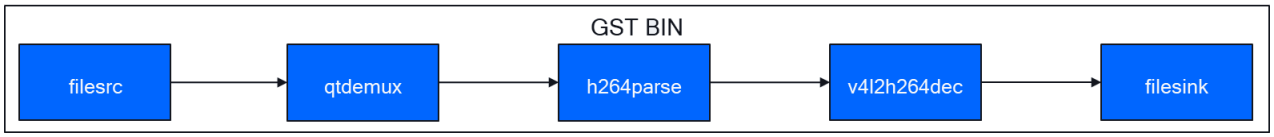


Figure 5-54 Pipeline to decode and save a stream to a file (H.264)

Single stream video playback (H.265)

- Decode and display a single stream (H.265):

```
gst-launch-1.0 -e filesrc location=/opt/<h265_file>.mp4 ! qtdemux !
queue ! h265parse ! v4l2h265dec capture-io-mode=4 output-io-mode=4 !
video/x-raw, format=NV12 ! waylandsink fullscreen=true
```

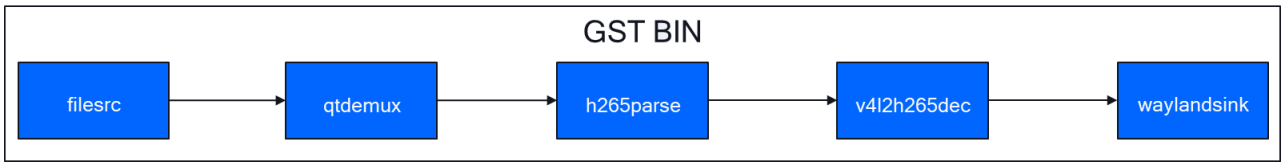


Figure 5-55 Pipeline to decode and display a single stream (H.265)

- Decode and save a stream to a file (H.265):

```
gst-launch-1.0 -e filesrc location=/opt/<h265_file>.mp4 ! qtdemux !
queue ! h265parse ! v4l2h265dec capture-io-mode=4 output-io-mode=4 !
video/x-raw, format=NV12 ! filesink location="/opt/video.yuv"
```

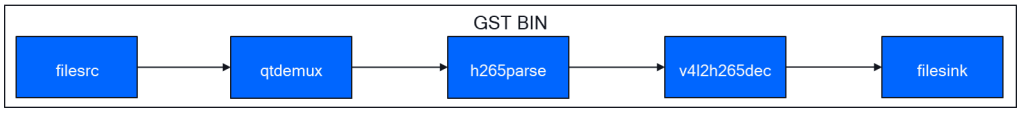


Figure 5-56 Pipeline to decode and save a stream to a file (H.265)

Single stream video playback (VP9)

The use cases show the playback of a 4K video stream from a file of media container format such as webm.

- Decode and display a single stream (VP9):

```
gst-launch-1.0 -e filesrc location=/opt/<vp9_file>.webm ! matroskademux !
v4l2vp9dec capture-io-mode=4 output-io-mode=4 ! video/x-raw, format=NV12 !
waylandsink fullscreen=true
```

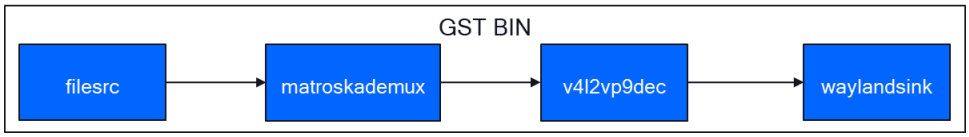


Figure 5-57 Pipeline to decode and display a single stream (VP9)

- Decode and save a stream to a file (VP9):

```
gst-launch-1.0 -e filesrc location=/opt/<vp9_file>.webm ! matroskademux !
v4l2vp9dec capture-io-mode=4 output-io-mode=4 ! video/x-raw, format=NV12 !
filesink  location="/opt/video.yuv"
```

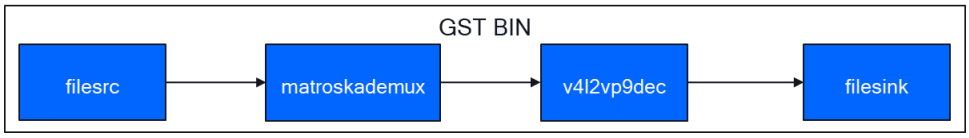


Figure 5-58 Pipeline to decode and save a stream to a file (VP9)

Two 1080p streams video playback simultaneously

The pipeline shows the simultaneous playback of two video streams from a file of a media container format such as MP4.

Known issue: In QCS9075, running these commands results in a blank screen, which blocks any further execution of the GST commands.

- In console 1:

```
gst-launch-1.0 -e filesrc location=/opt/<1080p_H264/AVC_file>.mp4 !
qtdemux ! queue ! h264parse ! v4l2h264dec capture-io-mode=4 output-io-
mode=4 ! video/x-raw, format=NV12 ! queue ! qtivcomposer name=mixer
sink_1::position="<480, 0>" sink_1::dimensions="<480, 270>" ! waylandsink
```

- In console 2:

```
gst-launch-1.0 -e filesrc location=/opt/<1080p_H264/AVC_file>.mp4 !
qtdemux ! queue ! h264parse ! v4l2h264dec capture-io-mode=4 output-io-
mode=4 ! video/x-raw, format=NV12 ! queue ! qtivcomposer name=mixer
sink_0::position="<0, 0>" sink_0::dimensions="<480, 270>" ! waylandsink
```

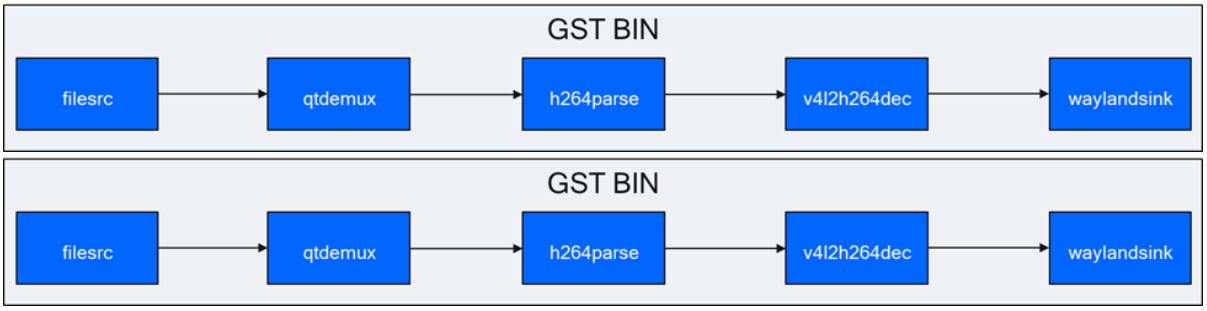


Figure 5-59 Simultaneous playback of two video streams from a file

Frame skip

The videorate plugin takes an incoming stream of timestamped video frames and produces a stream that matches the frame rate of the source pad.

- To perform a correction, drop and duplicate the frames.
- To force a conversion to a specific frame rate, use filtered caps on the source pad.

Use any of the following commands to run the use case:

- Single stream 1080p at 30 fps AVC file save:

```
gst-launch-1.0 -e qtiqmmfsrc name=camsrc video_0::type=video
video_1::type=preview ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1 ! queue !
v4l2h264enc capture-io-mode=4 output-io-mode=5 ! queue ! h264parse !
queue ! mp4mux ! queue ! filesink location=/opt/video.mp4
```

  Playback from a file stream at 15 fps for frame skip:

```
gst-launch-1.0 filesrc location="/opt/video.mp4" ! qtdemux ! queue !
h264parse ! v4l2h264dec capture-io-mode=4 output-io-mode=4 ! videorate !
video/x-raw, framerate=15/1 ! v4l2h264enc capture-io-mode=4 output-io-
```

```
mode=5 ! h264parse ! mp4mux ! queue ! filesink location="/opt/
video15fps.mp4"
```

- 1080p at 30 AVC → encode → decode → videorate at 20 fps → save to file:

```
gst-launch-1.0 -e qtiqmmfsrc ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1 ! queue !
v4l2h264enc capture-io-mode=4 output-io-mode=5 ! queue ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! videorate ! video/x-
raw,framerate=20/1 ! v4l2h264enc capture-io-mode=4 output-io-mode=5 !
h264parse ! mp4mux ! queue ! filesink location="/opt/video20fps.mp4"
```

RELATED INFORMATION

"Video playback" on page 157
"Video transformation using OpenCV" on page 178

5.2.6 Audio use cases

The audio GStreamer use cases show capture of audio from a file source and playback of the audio stream on a speaker.

Before you run the use cases, ensure that you complete the preconditions mentioned in Run multimedia use cases.

5.2.6.1 Audio (only) capture

The use cases show how to capture the PCM audio streams and store them in a file.

- Run all the commands in the SSH shell.
- To stop the use case, use **CTRL + C**.

Single PCM capture

```
gst-launch-1.0 -v pulsesrc volume=10 ! audioconvert ! wavenc ! filesink
location=/opt/<Audio_PCM>.wav
```

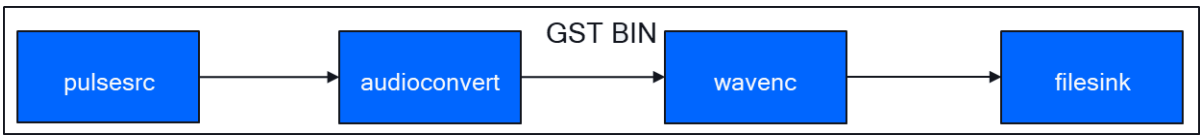


Figure 5-60 Pipeline for audio capture with PCM encode

Single FLAC capture

```
gst-launch-1.0 -v pulsesrc volume=10 ! audioconvert ! flacenc ! filesink
location=/opt/<flac_file>.flac
```

Figure 5-61 Pipeline for the audio capture with FLAC encode

5.2.6.2 Audio (only) playback

The pipelines show the use cases for playback of single audio streams on a speaker.

- Run all the commands in the SSH shell.
- To stop the use case, use **CTRL + C**.

Single PCM playback

```
gst-launch-1.0 -e filesrc location=/opt/<wav_file>.wav ! wavparse !
audioconvert ! pulsesink volume=10
```

The pipeline shows a single WAV stream taken from a file source and played on a speaker.

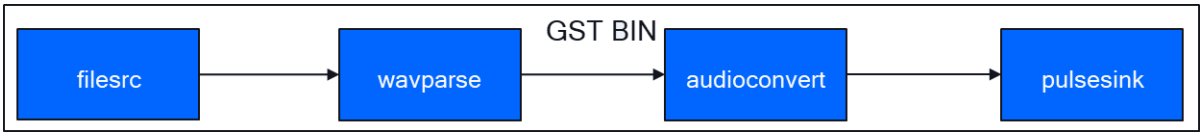


Figure 5-62 Pipeline for WAV stream playback

Single MP2/MP3 playback

- MP2

  ```
  gst-launch-1.0 -e filesrc location=/opt/<mp2_file>.mp2 ! mpegaudioparse !
  mpg123audiodec ! pulsesink volume=10
  ```

- MP3

  ```
  gst-launch-1.0 -e filesrc location=/opt/<mp3_file>.mp3 ! mpegaudioparse !
  mpg123audiodec ! pulsesink volume=10
  ```

The pipeline shows a single MP2 stream taken from a file source and played on a speaker.

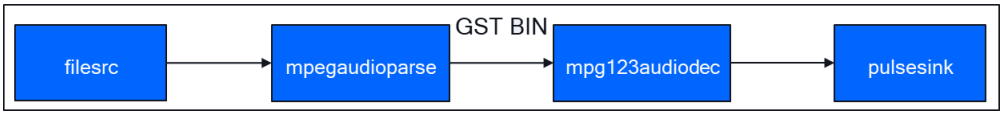


Figure 5-63 Pipeline for MP2/MP3 stream playback

Single FLAC playback

```
gst-launch-1.0 -e filesrc location=/opt/<flac_file>.flac ! flacparse !
flacdec ! pulsesink volume=10
```

The pipeline shows a single FLAC stream taken from a file source and played on a speaker.

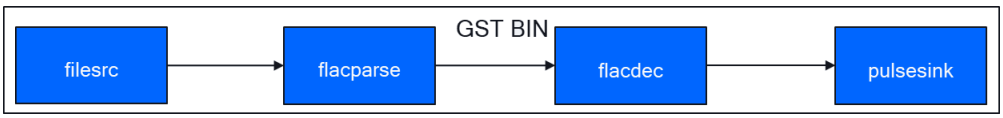


Figure 5-64 Pipeline for FLAC stream playback

Single OGG playback

```
gst-launch-1.0 -e filesrc location=/opt/<ogg_file>.ogg ! oggdemux !
vorbisdec ! audioconvert ! pulsesink volume=10
```

The pipeline shows a single OGG stream taken from a file source and played on a speaker.

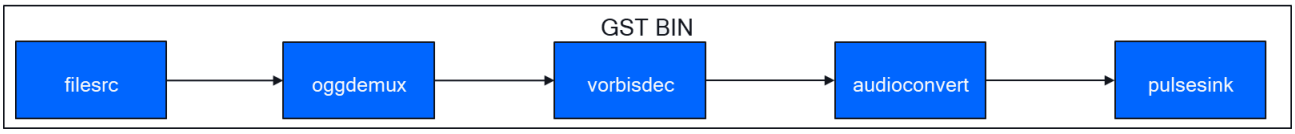


Figure 5-65 Pipeline for OGG stream playback

RELATED INFORMATION

"Audio playback" on page 162
"Audio recording" on page 164

5.2.7 Audio-video use cases

The audio-video (AV) use cases show AV capture and AV playback.

Before you run the use cases, ensure that you complete the preconditions mentioned in Run multimedia use cases.

5.2.7.1 Audio-video capture (H.264 video codec and MP3 audio codec)

The use cases show the AV capture where a 1080p stream from a camera is encoded as an H.264 stream, and sent to a multiplexer. The PCM `opt` is from an audio source is encoded as MP3, and sent to the multiplexer. The multiplexed files are then saved as an MP4 file.

1. Run the use case in the SSH shell:

   ```
   gst-launch-1.0 -e qtiqmmfsrc ! video/x-
   raw,format=NV12,width=1920,height=1080,framerate=30/1,interlace-
   mode=progressive,colorimetry=bt601 ! queue ! v4l2h264enc capture-io-mode=4
   output-io-mode=5 extra-
   controls="controls,video_bitrate=1000000,video_gop_size=29;" ! queue !
   h264parse ! muxer. pulsesrc do-timestamp=true provide-clock=false
   volume=10 ! audio/x-raw,format=S16LE,channels=1,rate=48000 !
   audioconvert ! queue ! lamemp3enc ! mp4mux name=muxer ! queue ! filesink
   location=/opt/1080p_AVC_MP3.mp4
   ```

2. Pull the recorded content to play it on the host computer:

   ```
   scp -r root@[DEVICE IP-ADDR]:/opt/1080p_AVC_PM3.mp4 user1@[HOST IP-ADDR]:/
   workspace
   ```

   You can play the camera stream from the MP4 file on a media player and the audio over the speakers.

To stop the use case, use **CTRL + C**.

The figure shows the flow of the pipeline execution:

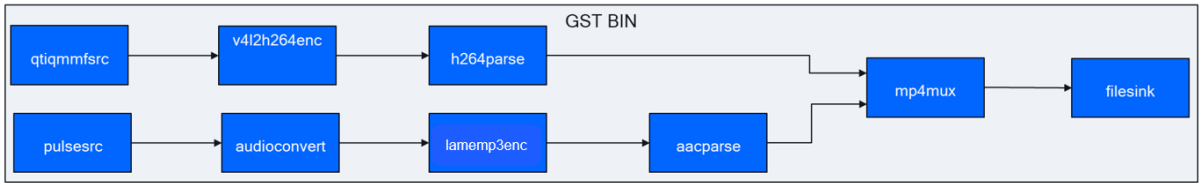


Figure 5-66 Pipeline for AV capture–camera source and audio source

5.2.7.2 Audio-video playback

The use case shows AV playback where the MP4 file is demultiplexed. The video stream is decoded and displayed. The audio stream is decoded and played over the speaker.

The following use cases describe a single AV playback with MP3 and FLAC formats.

- Run all the use cases in the SSH shell.
- To stop the use cases, use **CTRL + C**.

Single AV playback (AVC video + MP3 audio)

```
gst-launch-1.0 -e filesrc location=/opt/
<AV_file_with_H264_video_and_MP3_audio>.mp4 ! qtdemux name=demux demux. !
queue ! h264parse ! v4l2h264dec capture-io-mode=4 output-io-mode=4 !
waylandsink fullscreen=true demux. ! queue ! mpegaudioparse !
mpg123audiodec ! pulsesink volume=10
```

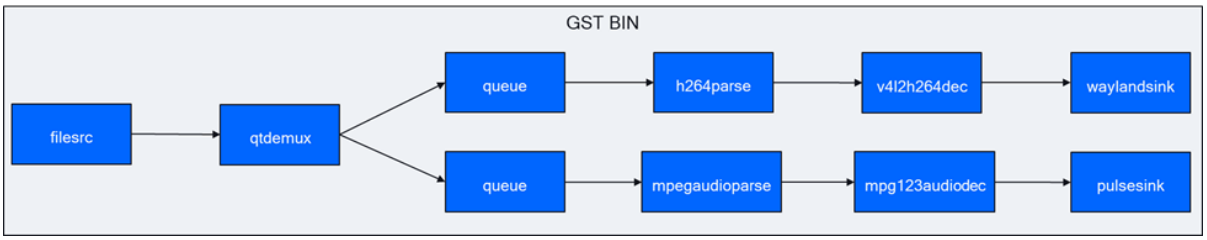


Figure 5-67 Pipeline for AV playback–AVC video + MP3 audio

Single AV playback (AVC video + FLAC audio)

```
gst-launch-1.0 -e filesrc location=/opt/
<AV_file_with_H264_video_and_FLAC_audio>.mp4 ! qtdemux name=demux demux. !
queue ! h264parse ! v4l2h264dec capture-io-mode=4 output-io-mode=4 !
waylandsink fullscreen=true demux. ! queue ! flacparse ! flacdec ! pulsesink
volume=10
```

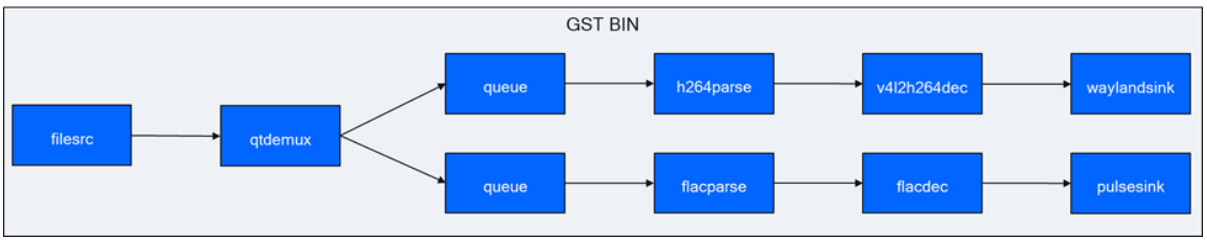


Figure 5-68 Pipeline for AV playback–AVC video + FLAC audio

RELATED INFORMATION

"Audio-video playback" on page 152

5.2.8 Video composition use cases

The video composition GStreamer use cases show how to read the video streams from the source, compose them, and then display them either side by side or picture-in-picture on a display device.

Before you run the use cases, ensure that you complete the preconditions mentioned in Run multimedia use cases.

5.2.8.1 16 480p offline stream side by side composition

The use case shows how to read 16 480p video streams from file sources and composing them side by side. These streams are then displayed as a single frame on a local display device.

Run the use case:

```
ulimit -n 4096 && gst-launch-1.0 qtivcomposer name=mix sink_0::position="<0,
0>" sink_0::dimensions="<480, 270>" sink_1::position="<480, 0>"
sink_1::dimensions="<480, 270>" sink_2::position="<960, 0>"
sink_2::dimensions="<480, 270>" sink_3::position="<1440, 0>"
sink_3::dimensions="<480, 270>" sink_4::position="<0, 270>"
sink_4::dimensions="<480, 270>" sink_5::position="<480, 270>"
sink_5::dimensions="<480, 270>" sink_6::position="<960, 270>"
sink_6::dimensions="<480, 270>" sink_7::position="<1440, 270>"
sink_7::dimensions="<480, 270>" sink_8::position="<0, 540>"
sink_8::dimensions="<480, 270>" sink_9::position="<480, 540>"
sink_9::dimensions="<480, 270>" sink_10::position="<960, 540>"
sink_10::dimensions="<480, 270>" sink_11::position="<1440, 540>"
sink_11::dimensions="<480, 270>" sink_12::position="<0, 810>"
sink_12::dimensions="<480, 270>" sink_13::position="<480, 810>"
sink_13::dimensions="<480, 270>" sink_14::position="<960, 810>"
sink_14::dimensions="<480, 270>" sink_15::position="<1440, 810>"
sink_15::dimensions="<480, 270>" mix. ! queue ! waylandsink sync=true
fullscreen=true filesrc location=/tmp/Animals_000_720p_180s_30FPS.mp4 !
qtdemux ! queue ! h264parse ! v4l2h264dec capture-io-mode=4 output-io-
mode=4 ! video/x-raw, format=NV12 ! queue ! mix. filesrc location=/tmp/
Animals_000_720p_180s_30FPS.mp4 ! qtdemux ! queue ! h264parse ! v4l2h264dec
capture-io-mode=4 output-io-mode=4 ! video/x-raw, format=NV12 ! queue ! mix.
filesrc location=/tmp/Animals_000_720p_180s_30FPS.mp4 ! qtdemux ! queue !
h264parse ! v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
```

```
format=NV12 ! queue ! mix. filesrc location=/tmp/
Animals_000_720p_180s_30FPS.mp4 ! qtdemux ! queue ! h264parse ! v4l2h264dec
capture-io-mode=4 output-io-mode=4 ! video/x-raw, format=NV12 ! queue ! mix.
filesrc location=/tmp/Animals_000_720p_180s_30FPS.mp4 ! qtdemux ! queue !
h264parse ! v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! queue ! mix. filesrc location=/tmp/
Animals_000_720p_180s_30FPS.mp4 ! qtdemux ! queue ! h264parse ! v4l2h264dec
capture-io-mode=4 output-io-mode=4 ! video/x-raw, format=NV12 ! queue ! mix.
filesrc location=/tmp/Animals_000_720p_180s_30FPS.mp4 ! qtdemux ! queue !
h264parse ! v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! queue ! mix. filesrc location=/tmp/
Animals_000_720p_180s_30FPS.mp4 ! qtdemux ! queue ! h264parse ! v4l2h264dec
capture-io-mode=4 output-io-mode=4 ! video/x-raw, format=NV12 ! queue ! mix.
filesrc location=/tmp/Animals_000_720p_180s_30FPS.mp4 ! qtdemux ! queue !
h264parse ! v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! queue ! mix. filesrc location=/tmp/
Animals_000_720p_180s_30FPS.mp4 ! qtdemux ! queue ! h264parse ! v4l2h264dec
capture-io-mode=4 output-io-mode=4 ! video/x-raw, format=NV12 ! queue ! mix.
filesrc location=/tmp/Animals_000_720p_180s_30FPS.mp4 ! qtdemux ! queue !
h264parse ! v4l2h264dec capture-io-mode=4 output-io-mode=4 ! queue ! mix.
filesrc location=/tmp/Animals_000_720p_180s_30FPS.mp4 ! qtdemux ! queue !
h264parse ! v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! queue ! mix. filesrc location=/tmp/
Animals_000_720p_180s_30FPS.mp4 ! qtdemux ! queue ! h264parse ! v4l2h264dec
capture-io-mode=4 output-io-mode=4 ! video/x-raw, format=NV12 ! queue ! mix.
filesrc location=/tmp/Animals_000_720p_180s_30FPS.mp4 ! qtdemux ! queue !
h264parse ! v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! queue ! mix. filesrc location=/tmp/
Animals_000_720p_180s_30FPS.mp4 ! qtdemux ! queue ! h264parse ! v4l2h264dec
capture-io-mode=4 output-io-mode=4 ! video/x-raw, format=NV12 ! queue ! mix.
filesrc location=/tmp/Animals_000_720p_180s_30FPS.mp4 ! qtdemux ! queue !
h264parse ! v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! queue ! mix.
```

To stop the use case, use **CTRL + C**.

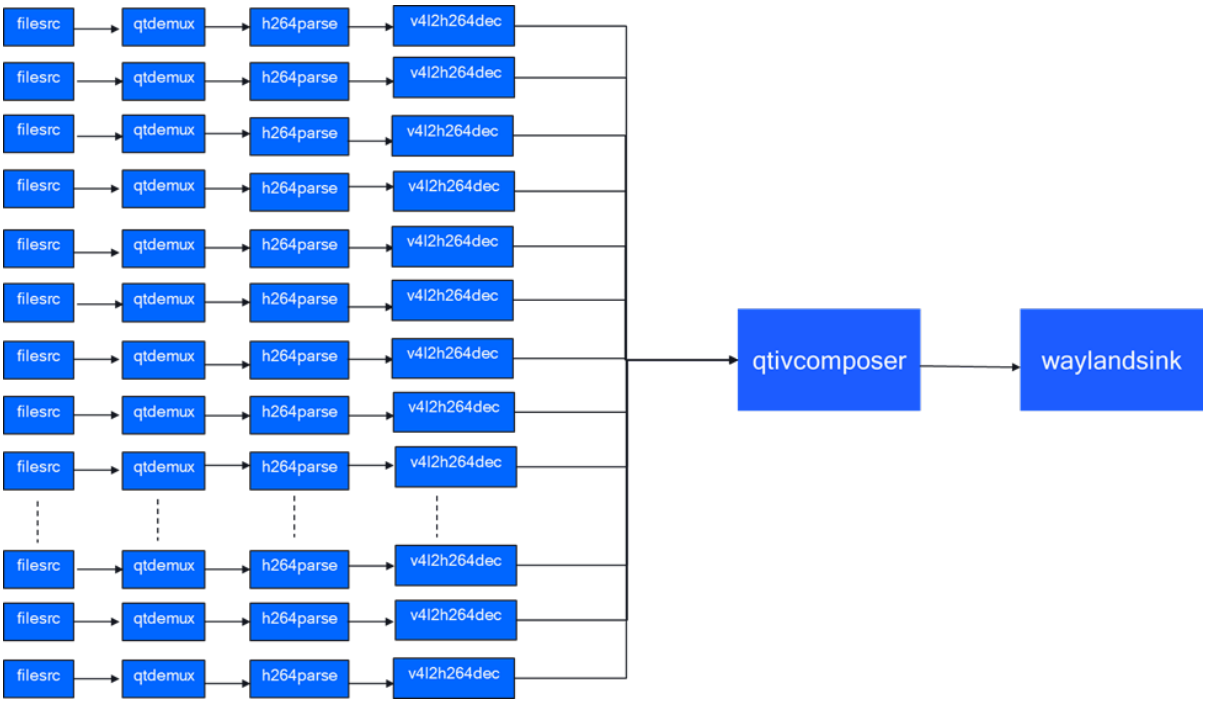


Figure 5-69 Pipeline for side by side display of 16 480p video streams–file sources

5.2.8.2 1080p + 1080p offline picture-in-picture composition

This use case shows the composition of two 1080p streams to get a picture-in-picture display. The two 1080p streams are read from two file sources.

Run the use case:

```
gst-launch-1.0 -e qtivcomposer name=mixer sink_0::position="<0, 0>"
sink_0::dimensions="<1280, 720>" sink_1::position="<590, 310>"
sink_1::dimensions="<640, 360>" mixer. ! queue ! waylandsink fullscreen=true
filesrc location=/tmp/Animals_000_720p_180s_30FPS.mp4 ! qtdemux ! queue !
h264parse ! v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! mixer. filesrc location=/tmp/Animals_000_720p_180s_30FPS.mp4 !
qtdemux ! queue ! h264parse ! v4l2h264dec capture-io-mode=4 output-io-
mode=4 ! video/x-raw, format=NV12 ! mixer.
```

To stop the use case, use **CTRL + C**.

Figure 5-70 Pipeline for picture-in-picture display of 1080p streams–file source

5.2.8.3 1080p live camera and 1080p offline picture-in-picture composition

The use case shows the composition of two 1080p streams to get a picture-in-picture display. The first 1080p stream is read from a file source. The second stream is collected from a camera.

Run the use case:

```
gst-launch-1.0 -e qtivcomposer name=mixer sink_0::position="<0, 0>"
sink_0::dimensions="<1280, 720>" sink_1::position="<590, 310>"
sink_1::dimensions="<640, 360>" mixer. ! queue ! waylandsink fullscreen=true
qtiqmmfsrc name=camsrc video_0::type=video video_1::type=preview ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1 ! mixer. filesrc
location=/opt/<1080p_H264/AVC_file>.mp4 ! qtdemux ! queue ! h264parse !
v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw, format=NV12 !
mixer.
```

To stop the use case, use **CTRL + C**.

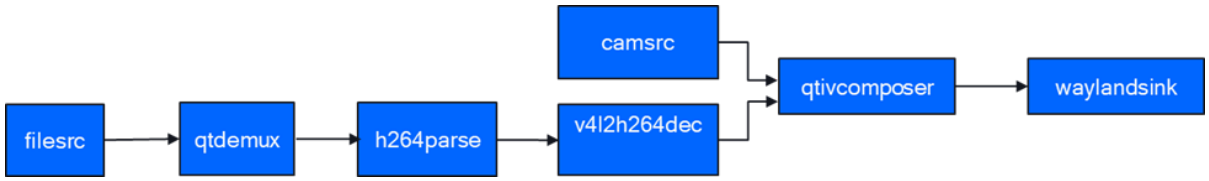


Figure 5-71 Pipeline for picture-in-picture display of 1080p streams–file source and camera source

RELATED INFORMATION

"Video composition" on page 170

5.2.9 Video overlay use cases

The pipelines show the use cases for video overlay. These use cases use the `dest-rect` property coordinates from the *overlay-mask* property of the qtivoverlay plugin.

Before you run the use cases, ensure that you complete the preconditions mentioned in Run multimedia use cases.

NOTE The commands in the following use cases use the 720p resolution. To change the resolution, update the respective `dest-rect` property coordinates in the command for that resolution. For more information, see qtivoverlay.

5.2.9.1 User text overlay

This use case overlays text on a video stream.

- Overlay on a video:

```
setprop persist.overlay.use_c2d_blit 2
```

```
gst-launch-1.0 filesrc location=/opt/Animals_000_720p_180s_30FPS.mp4 !
qtdemux ! queue ! h264parse ! v4l2h264dec capture-io-mode=4 output-io-
mode=4 ! qtioverlay overlay-text="text0, text=\"Qualcomm\ Intelligence\",
```

```
color=(uint)0xFFFF00FF, dest-rect=<160, 624, 944, 50>;" ! waylandsink
fullscreen=true
```

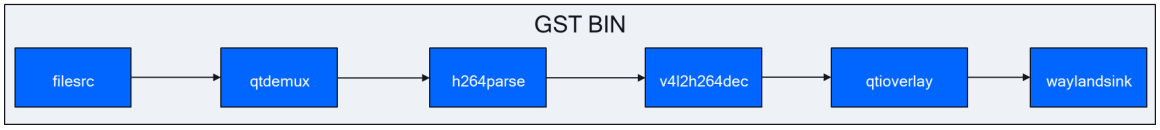


Figure 5-72 Pipeline for text overlay on video–file source

- Overlay on a live stream from a camera:

```
setprop persist.overlay.use_c2d_blit 2
```

```
gst-launch-1.0 -e qtiqmmfsrc name=camsrc video_0::type=video
video_1::type=preview ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1 ! qtioverlay overlay-
text="text0, text=\"Qualcomm\ Intelligence\", color=(uint)0xFFFF00FF, dest-
rect=<160, 624, 944, 50>;" ! waylandsink fullscreen=true
```

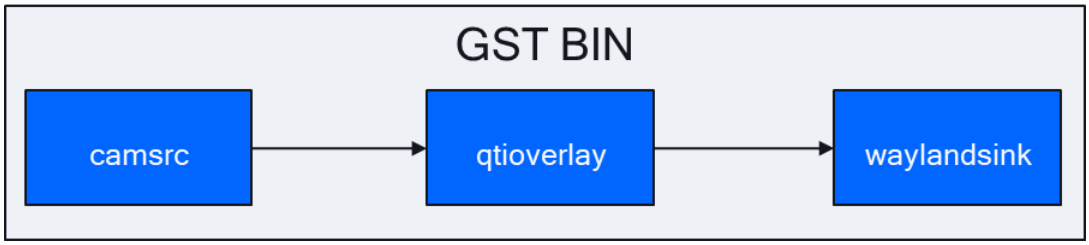


Figure 5-73 Pipeline for text overlay on video–camera source

5.2.9.2 Date and time overlay

This use case shows the date (in MMDDYYYY) and time (in HHMMSS) to be overlaid on the video stream.

- Overlay on a video:

```
setprop persist.overlay.use_c2d_blit 2
```

```
gst-launch-1.0 filesrc location=/opt/<H264_file>.mp4 ! qtdemux ! queue !
h264parse ! v4l2h264dec capture-io-mode=4 output-io-mode=4 ! qtioverlay
overlay-date="date0, date-format=\"MMDDYYYY\", time-format=\"HHMMSS_24HR
\", color=(uint)0xFF00FFFF, dest-rect=<400, 400, 256, 80>;" ! waylandsink
fullscreen=true
```

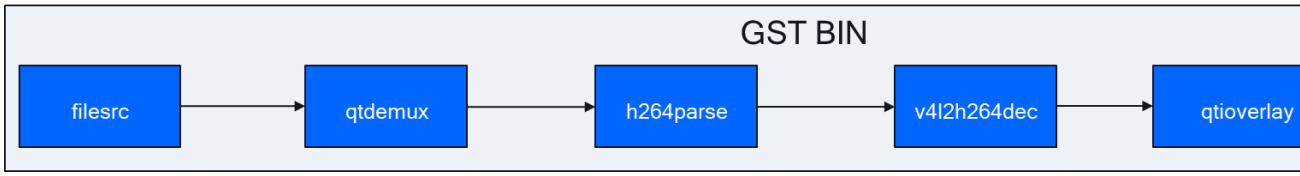


Figure 5-74 Pipeline for text date and time overlay on video–file source

- Overlay on a live stream from a camera:

```
setprop persist.overlay.use_c2d_blit 2
```

```
gst-launch-1.0 -e qtiqmmfsrc name=camsrc video_0::type=video
video_1::type=preview ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1 ! qtioverlay overlay-
date="date0, date-format=\"MMDDYYYY\", time-format= \"HHMMSS_24HR\",
color=(uint)0xFF00FFFF, dest-rect=<400, 400, 256, 80>;" ! waylandsink
fullscreen=true
```

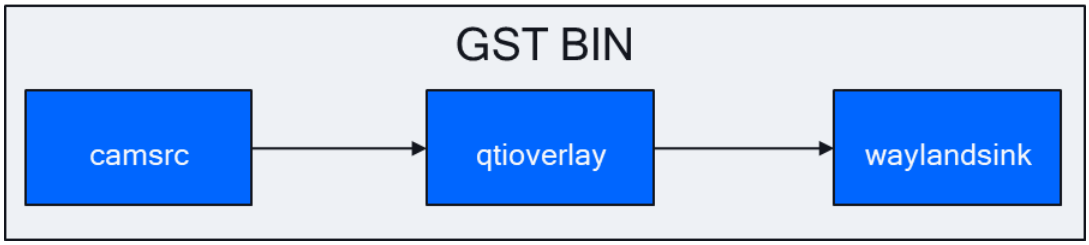


Figure 5-75 Pipeline for text date and time overlay on video–camera source

5.2.9.3 Static image overlay

This use case shows how to overlay a static image on a video stream.

1. Push the static image to the device:

```
scp -r <static_image>.rgba  root@[DEVICE IP-ADDR]:/opt
```

2. Run the use cases:

   - Overlay on a video:

```
setprop persist.overlay.use_c2d_blit 2
```

```
gst-launch-1.0 filesrc location=/opt/<H264_file>.mp4 ! qtdemux !
queue ! h264parse ! v4l2h264dec capture-io-mode=4 output-io-mode=4 !
qtioverlay overlay-simg="image0, image=\"/opt/overlay_test_464_109.rgba
\", resolution=<464, 109>, dest-rect=<140, 130, 464, 109>;" !
waylandsink fullscreen=true
```

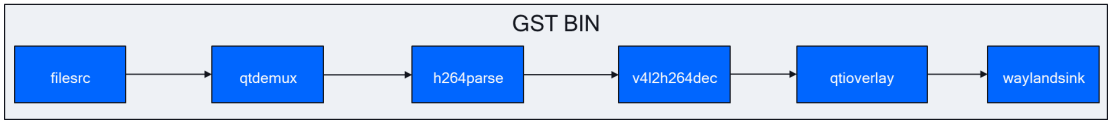


Figure 5-76 Pipeline for static image overlay on video–file source

   - Overlay on alive stream from a camera:

```
setprop persist.overlay.use_c2d_blit 2
```

```
gst-launch-1.0 -e qtiqmmfsrc name=camsrc video_0::type=video
video_1::type=preview ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1,compression=ubwc !
 qtioverlay overlay-simg="image0, image=\"/opt/overlay_test_464_109.rgba
\", resolution=<464, 109>, dest-rect=<140, 130, 464, 109>;" !
waylandsink fullscreen=true
```

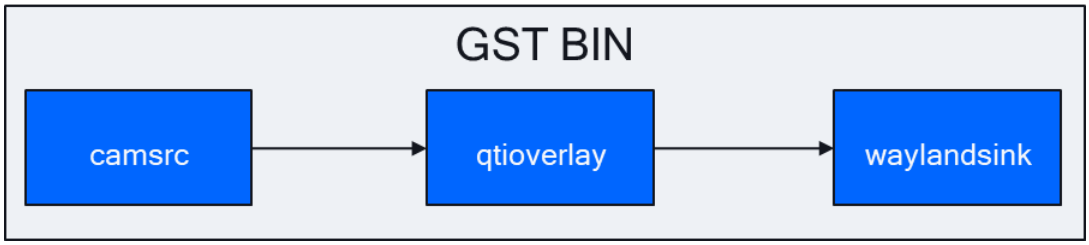


Figure 5-77 Pipeline for static image overlay on video–camera source

5.2.9.4 Bounding box overlay

This use case shows how to overlay a bounding box with a label on the video stream.

- Overlay on a video:

```
setprop persist.overlay.use_c2d_blit 2
```

```
gst-launch-1.0 -e qtiqmmfsrc name=camsrc video_0::type=video
video_1::type=preview ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1,compression=ubwc !
qtioverlay overlay-simg="image0, image=\"/opt/overlay_test_464_109.rgba\",
resolution=<464, 109>, dest-rect=<140, 130, 464, 109>;" ! waylandsink
fullscreen=true
```

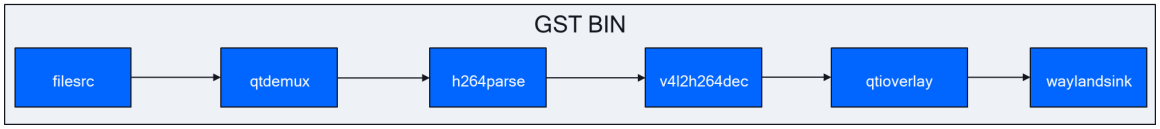


Figure 5-78 Pipeline for bounding box overlay on video–file source

- Overlay on a live stream from a camera:

```
setprop persist.overlay.use_c2d_blit 2
```

```
gst-launch-1.0 -e qtiqmmfsrc name=camsrc video_0::type=video
video_1::type=preview ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1,compression=ubwc !
qtioverlay overlay-simg="image0, image=\"/opt/overlay_test_464_109.rgba\",
resolution=<464, 109>, dest-rect=<140, 130, 464, 109>;" ! waylandsink
fullscreen=true
```

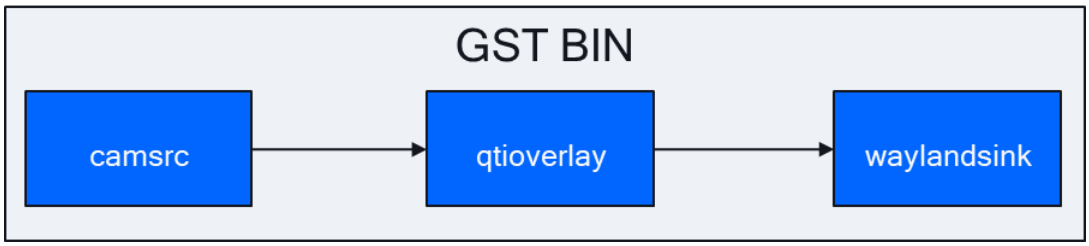


Figure 5-79 Pipeline for bounding box overlay on video–camera source

5.2.9.5 Inverse privacy mask

This use case shows how to overlay the inverse privacy mask on the video stream.

The inverse privacy mask is applied on all other regions except the *circle* in the *overlay-mask* property.

- Overlay on a video:

```
setprop persist.overlay.use_c2d_blit 2
```

```
gst-launch-1.0 -e qtiqmmfsrc name=camsrc video_0::type=video
video_1::type=preview ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1,compression=ubwc !
qtioverlay overlay-simg="image0, image=\"/opt/overlay_test_464_109.rgba\",
resolution=<464, 109>, dest-rect=<140, 130, 464, 109>;" ! waylandsink
fullscreen=true
```

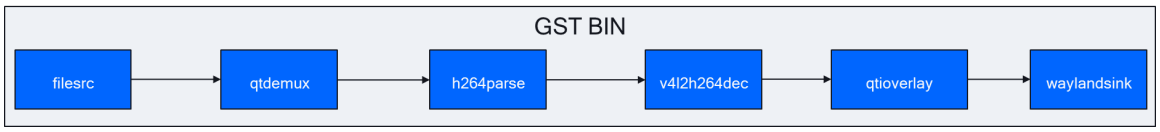


Figure 5-80 Pipeline for inverse privacy mask overlay on video–file source

- Overlay on a live stream from a camera:

```
setprop persist.overlay.use_c2d_blit 2
```

```
gst-launch-1.0 -e qtiqmmfsrc ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1 ! qtioverlay overlay-
mask="mask0, circle=<960, 540, 400>, inverse=true, dest-rect=<0, 0, 1920,
1080>, color=0x202020FF" ! waylandsink fullscreen=true
```

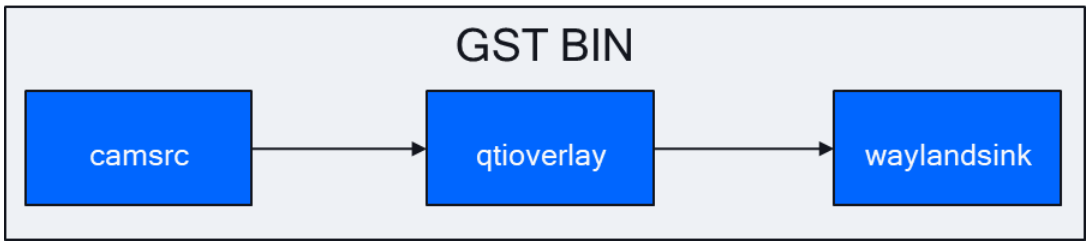


Figure 5-81 Pipeline for inverse privacy mask overlay on video–camera source

5.2.9.6 Non-inverse privacy mask

This use case shows how to overlay the non-inverse privacy mask on the video stream.

- Overlay on a video:

```
setprop persist.overlay.use_c2d_blit 2
```

```
gst-launch-1.0 filesrc location=/opt/<input_file>.mp4 ! qtdemux ! queue !
h264parse ! v4l2h264dec capture-io-mode=4 output-io-mode=4 ! qtioverlay
overlay-mask="mask0, circle=<640, 360, 200>, dest-rect=<0, 0, 1280, 720>,
color=0x202020FF" ! waylandsink fullscreen=true
```

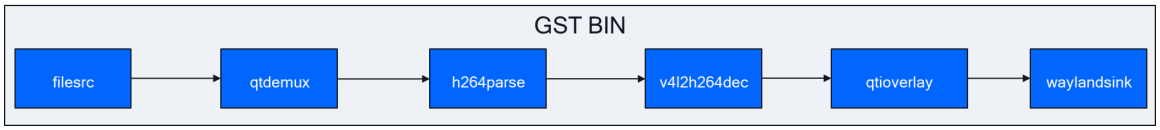


Figure 5-82 Pipeline for non-inverse privacy mask overlay on video–file source

- Overlay on a live stream from a camera:

```
setprop persist.overlay.use_c2d_blit 2
```

```
gst-launch-1.0 -e qtiqmmfsrc ! video/x-raw,format=NV12,width=1920,height=1080,framerate=30/1 ! qtioverlay overlay-mask="mask0, circle=<960, 540, 400>, dest-rect=<0, 0, 1920, 1080>, color=0x202020FF" ! waylandsink fullscreen=true
```

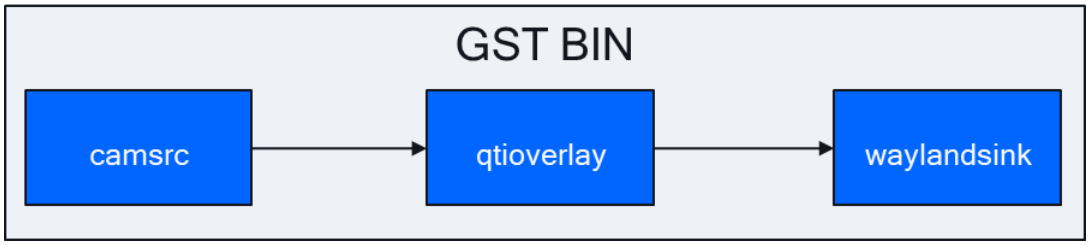


Figure 5-83 Pipeline for non-inverse privacy mask overlay on video–camera source

5.2.9.7 Polygon privacy mask

A polygon privacy mask allows you to create privacy zones on an image using a polygonal shape. Using a polygon allows you to make the masking smoother and more exact.

- The polygon privacy mask is applied to the region enclosed within the polygon.
- The inverse polygon privacy mask is applied to all regions except the one enclosed within the polygon.

Before you begin, configure the display:

```
export XDG_RUNTIME_DIR=/dev/socket/weston && export WAYLAND_DISPLAY=wayland-1
```

```
setprop persist.overlay.use_c2d_blit 2
```

Enable polygon privacy mask

- File source:

```
gst-launch-1.0 filesrc location=/opt/<input_file>.mp4 ! qtdemux ! queue ! h264parse !  v4l2h264dec capture-io-mode=4 output-io-mode=4 ! qtioverlay overlay-mask="mask0, polygon=<5, <360, 640, 780, 500, 300>, <360, 540, 180, 180, 100>>, dest-rect=<0, 0, 1280, 720>, color=0x202020FF" ! waylandsink fullscreen=true
```

- Camera:

```
gst-launch-1.0 -e qtiqmmfsrc ! video/x-raw,format=NV12,width=1920,height=1080,framerate=30/1 ! qtioverlay overlay-mask="mask0, polygon=<5, <360, 640, 780, 500, 300>, <360, 540, 180, 180,
```

```
100>>, dest-rect=<0, 0, 1920, 1080>, color=0x202020FF" ! waylandsink
fullscreen=true
```

Enable inverse polygon privacy mask

- File source:

```
gst-launch-1.0 filesrc location=/opt/<input_file>.mp4 ! qtdemux ! queue !
h264parse ! v4l2h264dec capture-io-mode=4 output-io-mode=4 ! qtioverlay
overlay-mask="mask0, polygon=<3, <360, 640, 780>, <360, 540, 180>>,
inverse=true, dest-rect=<0, 0, 1280, 720>, color=0x202020FF" ! waylandsink
fullscreen=true
```

- Camera:

```
gst-launch-1.0 -e qtiqmmfsrc ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1 ! qtioverlay overlay-
mask="mask0, polygon=<3, <360, 640, 780>, <360, 540, 180>>, inverse=true,
dest-rect=<0, 0, 1920, 1080>, color=0x202020FF" ! waylandsink
fullscreen=true
```

Update polygon privacy mask property at runtime

1. Set the privacy mask:

```
gst-pipeline-app -e qtiqmmfsrc ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1 ! qtioverlay overlay-
mask="mask0, polygon=<7, <360, 640, 780, 640, 500, 360, 180>, <360, 360,
180, 120, 200, 120, 180>>, inverse=true, dest-rect=<0, 0, 1920, 1080>,
color=0x202020FF" ! waylandsink fullscreen=true
```

2. In the pipeline controls, select `Playing mode - 3`.
3. To change the property in the pipeline controls, select `Plugin mode - p`.
4. From the plugins, select `overlay - 2`.
5. In the plugin properties, select `overlay-mask - 4`.
6. Add the following value:

```
mask0, polygon=<5, <360, 640, 780, 500, 300>, <360, 540, 180, 180, 100>>,
inverse=true, dest-rect=<0, 0, 1920, 1080>, color=0x202020FF
```

5.2.10 Video split use cases

This use case shows how to split one video into nine streams and then perform the composition.

1. Complete the preconditions mentioned in Run multimedia use cases.
2. Run the use case:

```
gst-launch-1.0 filesrc location=/opt/<input_file>.mp4 ! qtdemux ! queue !
h264parse ! v4l2h264dec capture-io-mode=4 output-io-mode=4 ! video/x-raw,
format=NV12 ! queue ! qtivsplit name=split \
split. ! queue ! mixer. \
split. ! queue ! mixer. \
```

```
split. ! queue ! mixer. \
split. ! queue ! mixer. \
split. ! queue ! mixer. \
split. ! queue ! mixer. \
split. ! queue ! mixer. \
split. ! queue ! mixer. \
split. ! queue ! mixer. \
qtivcomposer name=mixer sink_0::position="<0, 0>"
sink_0::dimensions="<640, 360>" \
sink_1::position="<640, 0>" sink_1::dimensions="<640, 360>" \
sink_2::position="<1280, 0>" sink_2::dimensions="<640, 360>" \
sink_3::position="<0, 360>" sink_3::dimensions="<640, 360>" \
sink_4::position="<640, 360>" sink_4::dimensions="<640, 360>" \
sink_5::position="<1280, 360>" sink_5::dimensions="<640, 360>" \
sink_6::position="<0, 720>" sink_6::dimensions="<640, 360>" \
sink_7::position="<640, 720>" sink_7::dimensions="<640, 360>" \
sink_8::position="<1280, 720>" sink_8::dimensions="<640, 360>" mixer. !
queue ! waylandsink fullscreen=true
```

The figure shows the flow of the pipeline execution:

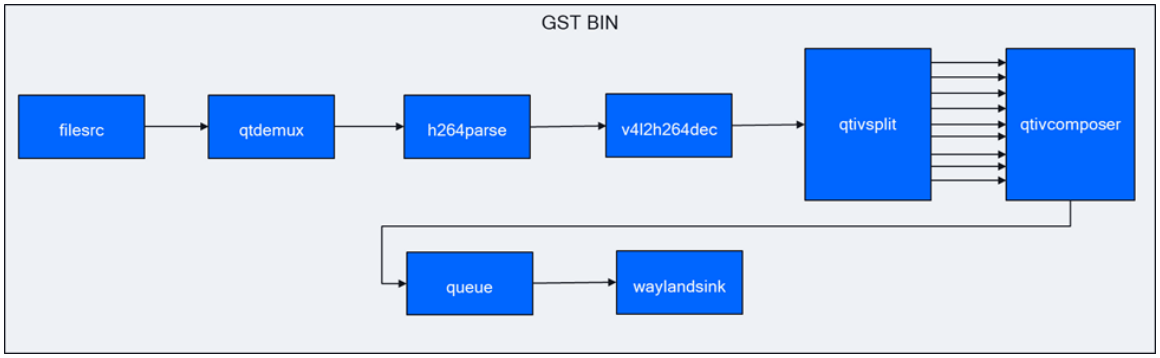


Figure 5-84 Pipeline for qtivsplit composing 9 1080p streams from one input video file

6 Qualcomm GStreamer architecture and plugins

The GStreamer plugins interact with Qualcomm IM SDK multimedia and AI/ML components to build the multimedia and AI/ML use cases.

6.1 GStreamer plugin architecture

The Qualcomm IM SDK hides the complexity of the hardware within the plugin architecture and provides APIs to applications. Using this framework, you can create applications without the need to access the low-level platform libraries and hardware details, which can vary across platforms.

The Configure Qualcomm GStreamer plugins include video encoding/decoding, camera ISP, GPU, display, audio DSP (aDSP), and AI/ML accelerators.

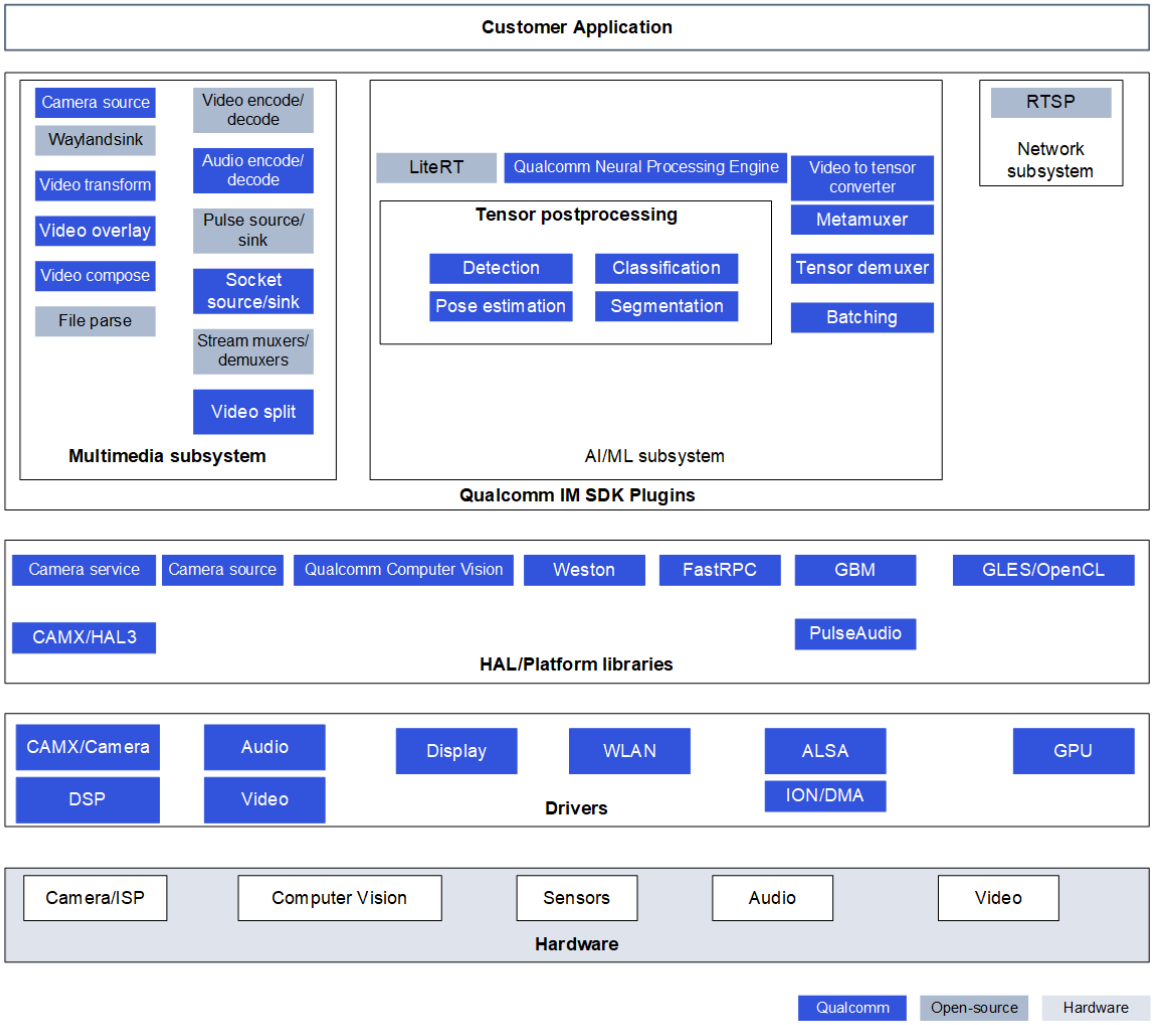


Figure 6-1 Qualcomm IM SDK GStreamer plugin architecture

6.1.1 Graphics and display architecture

The Wayland server uses the Wayland protocol to enable communication between the display and graphics servers, and its clients. The display and graphics architecture uses the Weston server (Weston).

Weston manages both composition and display while it runs as a separate process in the system.

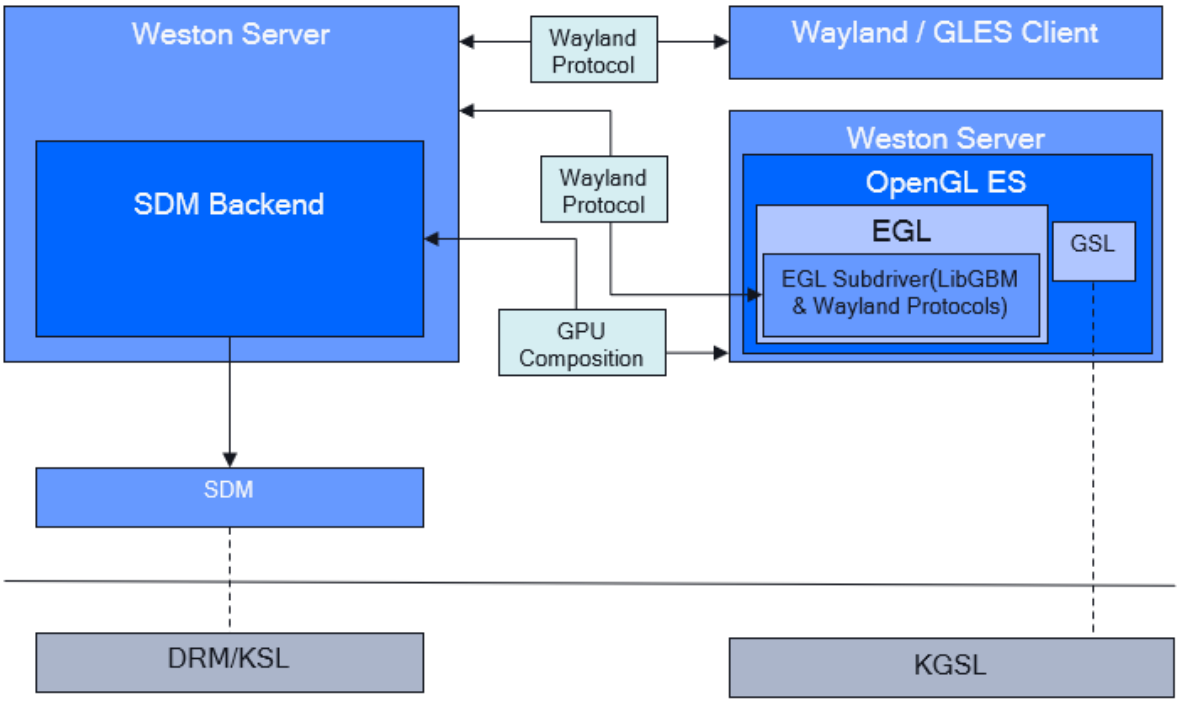


Figure 6-2 Weston/Wayland architecture

Table 6-1 Weston/Wayland components

Component	Description
Wayland/ GLES client	Implements the Wayland protocol for Weston.
Weston server	Implements the Wayland compositor. ▪ Uses the Kernel Mode Setting (KMS) to set up the display. ▪ Performs the compositing using OpenGL ES and Direct Rendering Manager (DRM).
Secure dynamic messaging (SDM) back-end	▪ Uses display hardware abstraction layer (HAL) to interact with the display hardware. ▪ Provides several platform-dependent implementations with one of it designated for DRM and KMS.
GBM buffer management library (libGBM)	▪ Includes DMA back-end. ▪ Used for zero-copy buffer sharing between display and graphics.
EGL platform-specific driver (EGL sub-driver)	Interacts with GBM and Wayland protocol to communicate with the Weston compositor.

6.1.2 Camera architecture

The camera architecture provides a high-level abstraction on how the underlying camera driver and hardware interact with the qtiqmmfsrc camera plugin.

The figure shows the camera architecture, where the qtimmfsrc plugin, acts as a client to the Qualcomm Camera Service.

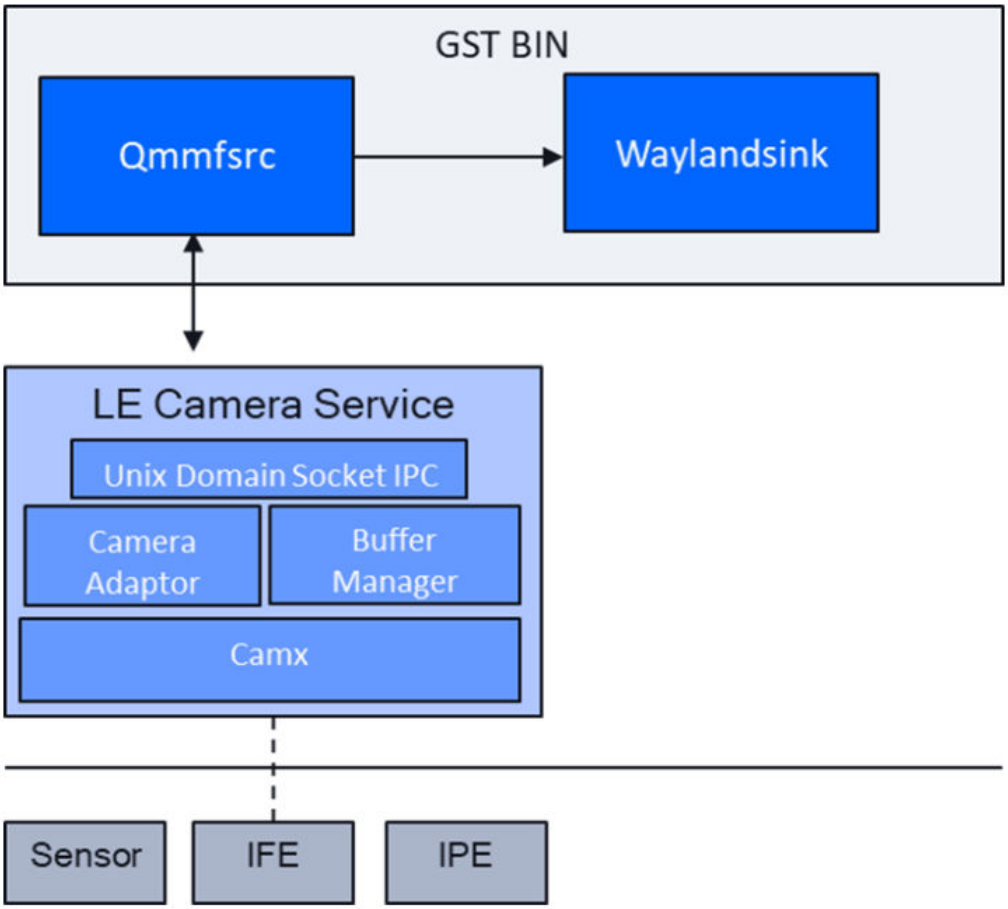


Figure 6-3 Camera pipeline

Table 6-2 Camera pipeline components

Component	Description
Linux embedded (LE) camera service	▪ Runs as a daemon in the system: ▫ Provides easy remote procedure call (RPC) APIs to control the camera. ▫ Exposes helper client APIs, which perform RPC between client and server. ▪ Uses the HAL3 API, which interacts with the camera back-end (CamX) and camera driver to configure camera sensor and image signal processor (ISP) hardware. ▪ Uses GBM to assign buffers for each camera stream and submits them to HAL3.
Client/server architecture	Allows the multi-client and multi-camera use cases to do the following:

Table 6-2 Camera pipeline components (cont.)

Component	Description
	■ Create many instances of qtiqmmfsrc where one instance of the plugin corresponds to one physical or logical camera. ■ Implement camera use cases. These instances can be in the same process or different processes depending upon the use case
qtiqmmfsrc	■ Provides several streams in parallel (many source pads), and each stream can be of different formats (NV12/21 or MJPEG). For example, if there are three parallel streams, they're processed as follows: a. One stream for local storage. b. One stream for YUV stream for live camera preview on a local display. c. One for network streaming. ■ Supports ‘N’ number of parallel streams, where 'N' depends on camera ISP capability.
Waylandsink	■ Uses the raw YUV (NV12/21) stream output to render the camera frames to a physical display. ■ Achieves a live camera preview use case. ■ Allows the postprocessing element to improve the quality. ■ Allows the ML inferencing elements to do inferencing on a live camera.
Buffers	■ The same buffers are circulated to the clients (qtiqmmfsrc) and other plugins in the pipeline. ■ The buffer pools manage the buffers to avoid per-frame allocation.

RELATED INFORMATION

“Camera” on page 267

6.1.3 Video architecture

The v4l2h264enc and v4l2h265enc video encode plugins connect with the camera plugin to implement the advance video encode (H.264 or H.265) use cases. The v4l2h264dec and v4l2h265dec video decode plugins connect with Waylandsink to implement the video playback use cases.

Encode

The figure shows the architecture for video encoding. The v4l2h264enc and v4l2h265enc plugins are connected to a file multiplexer (MP4 or MPEGTS) to save the encode bitstream to the file system.

For video encode use cases, see Video encode and decode.

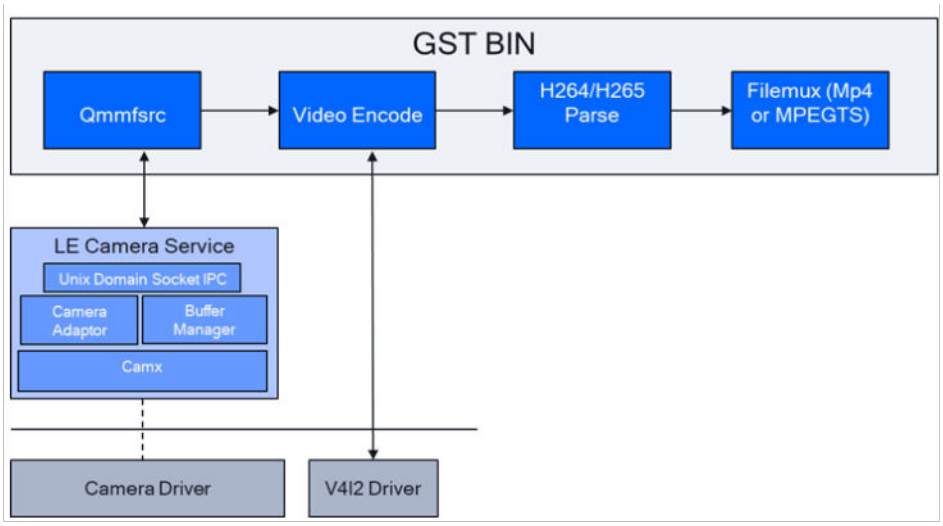


Figure 6-4 Video encode pipeline

Table 6-3 Video encode components

Components	Description
qtiqmmfsrc	Captures and provides several video streams in parallel. For more information about the camera service, see Camera architecture.
Video encode plugins	Uses the V4L2 driver and APIs to encode the video stream. ▪ v4l2h264enc: Encodes the video for AVC (H.264) format. ▪ v4l2h265enc: Encodes the video for HEVC (H.265) format.
H.264/H.265 parse	Parses the video stream.
Filemux	Multiplexes encoded video stream and saves to a file.

Decode

The figure shows the architecture for video decoding where the video stream from a file is demultiplexed, decoded, and displayed on a local device.

The decode parameters are exposed as a property to the application to provide control over the video decode pipeline.

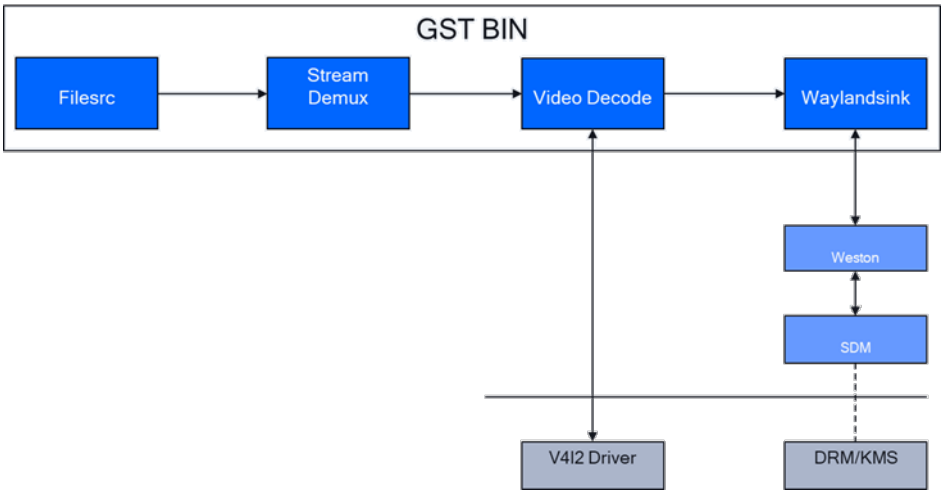


Figure 6-5 Video decode pipeline

For video decode use cases, see Video playback use cases.

Table 6-4 Video decode components

Components	Description
Video decode plugins	Uses the V4L2 driver and APIs to decode the video stream and connects with the Waylandsink plugin. ■ v4l2h264dec: Decodes the video for AVC (H.264) format. ■ v4l2h265dec: Decodes the video for HEVC (H.265) format.
Waylandsink	1. Receives the GBM buffers (decoded buffers) as an input. 2. Sends them to the Weston server (through Wayland protocol) for display composition.

6.1.4 Audio architecture

For audio capture and playback, use the pulsesrc and pulsesink GStreamer plugins. You can encode and decode audio through an open-source software.

Capture

The figure shows the audio capture using the pulsesrc plugin. The audio stream is then processed and written to a file.

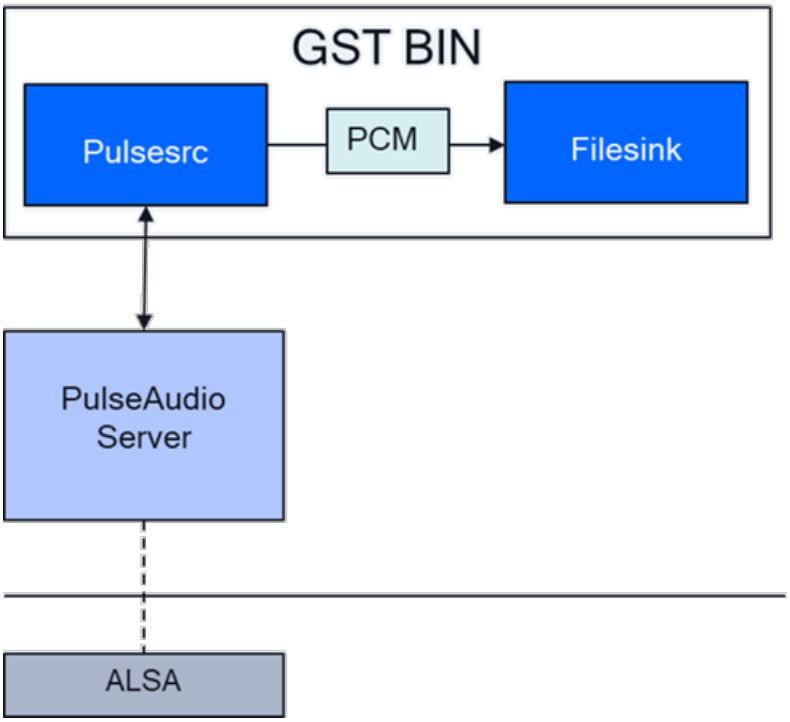


Figure 6-6 Audio capture pipeline

Table 6-5 Audio capture components

Component	Description
Pulsesrc	▪ Captures the audio and shares it with the underlying PulseAudio server. ▪ Sets the audio source from which the pulse-code modulation (PCM) audio samples are captured. ▪ Saves the PCM samples to a file.
PulseAudio server	▪ Interacts with the Qualcomm hardware. ▪ Uses a pluggable module to interact with the ALSA driver to process the audio data.

Playback

Use the pulsesink plugin for audio playback. It's an upstream plugin that allows you to play audio from various audio sources (live source or an encoded audio file).

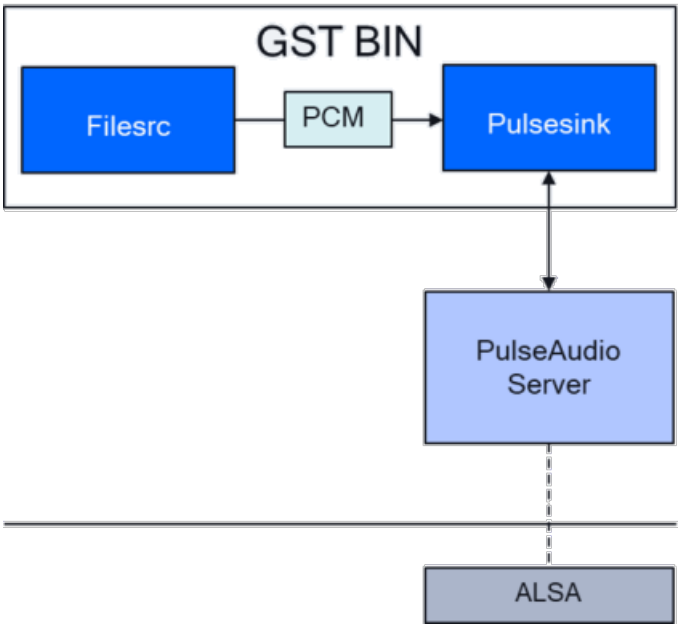


Figure 6-7 Audio playback pipeline

Table 6-6 Audio playback components

Component	Description
Pulsesink	▪ Interacts with the underlying PulseAudio server. ▪ Sets the audio playback sink on which the audio is played. ▪ Plays only the PCM data.
PulseAudio server	▪ Interacts with the Qualcomm hardware. ▪ Uses a pluggable module to interact with the ALSA driver to process the audio data for playback.

Encode

You can encode the audio using an open-source audio encoder plugin.

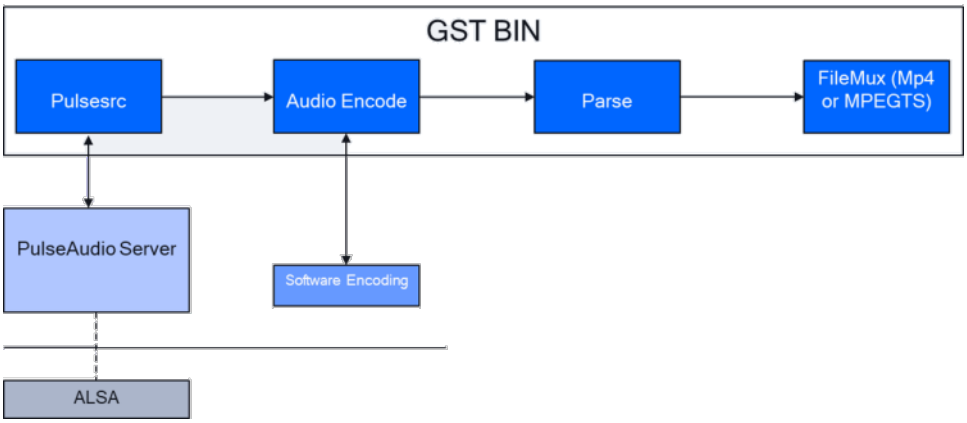


Figure 6-8 Audio encoding pipeline

Table 6-7 Audio encoding components

Component	Description
Pulsesrc	Captures the audio and shares it with the underlying PulseAudio server.
PulseAudio server	Interacts with the ALSA driver to process the audio data for encoding.
Encode	Encodes the audio data using an open-source software.
Parse	Parses the audio data.
Filemux	Multiplexes it and saves it in an MP4 or MPEGTS container.

Decode

You can decode the audio using an open-source audio decoder plugin.

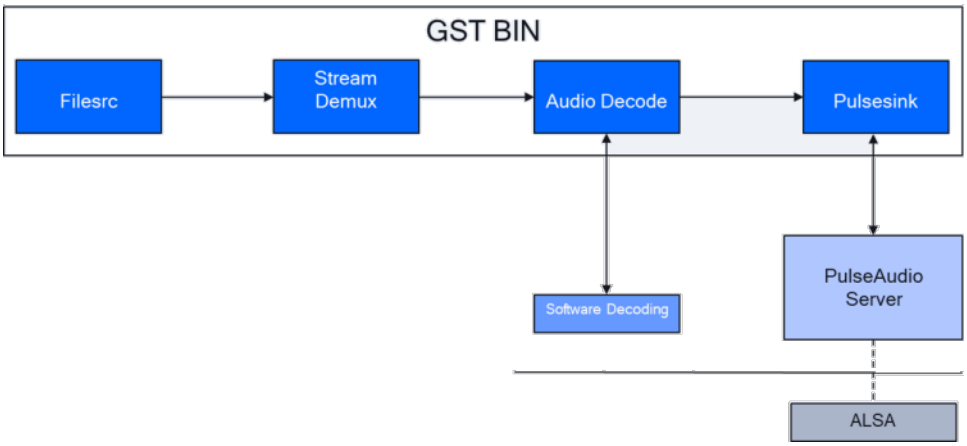


Figure 6-9 Audio decode pipeline

Table 6-8 Audio decode components

Component	Description
Filesrc	Reads the audio data.
Stream demux	Demultiplexes the audio data.
Decode	Decodes the audio data using an open-source software.
Pulsesink	Interacts with the underlying PulseAudio server and plays the decoded audio.
PulseAudio server	Interacts with the ALSA driver to process the audio data for decoding.

6.1.5 Machine learning architecture

Qualcomm IM SDK supports machine learning use cases that include video preprocessing, model inference, output tensor postprocessing, and inference result overlay to stream.

The machine learning framework provides the following types of video analytics:

- Image classification
- Object detection
- Image segmentation

With machine learning plugins in the framework, use the following for inferencing:

- LiteRT (or TFLite)
- Qualcomm Neural Processing SDK

For each of the engines, the delegates speed up the model inference performance.

ML pipeline processing

The figure depicts the pipeline in which the captured video/audio stream is processed among the preprocessing, inference, and postprocessing plugins.

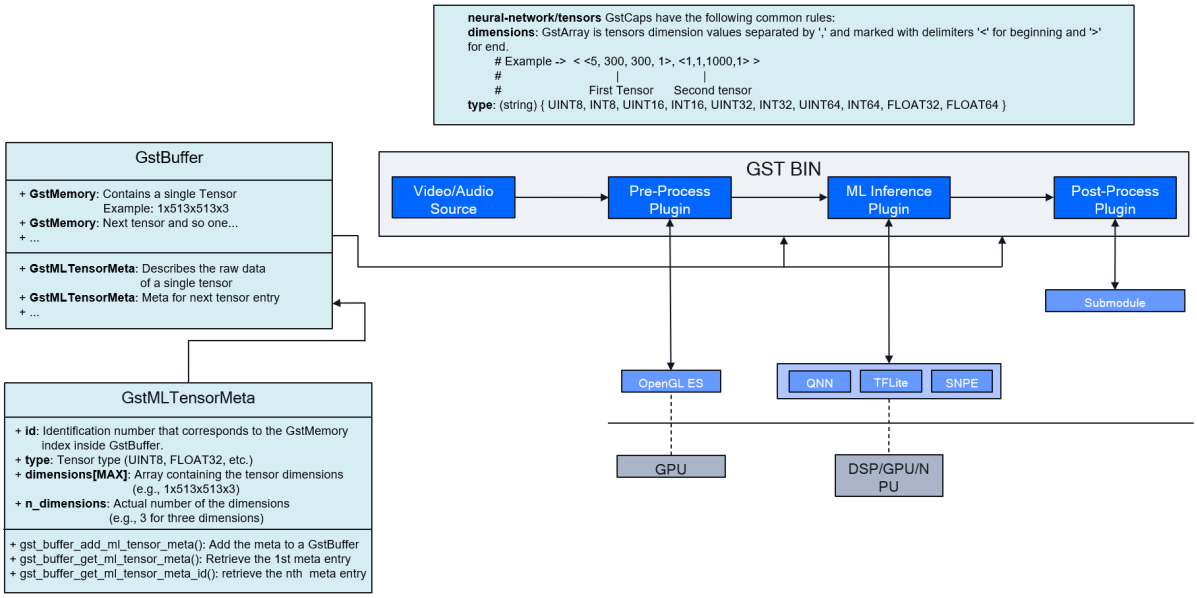


Figure 6-10 ML pipeline

The table lists the sequential stages during which the video/audio stream is captured, processed, and the output is either displayed on a screen or saved to a file.

Table 6-9 ML pipeline processing stages

Process	Description
Source	▪ qtiqmmfsrc captures the YUV streams from live camera source. ▪ The filesrc plugin captures any format from an offline video that requires format conversion.
Preprocessing	qtimlvconverter does the following to prepare the video stream for inferencing: ▪ Color conversion ▪ Resize ▪ Mean subtraction
Inferencing	1. The preprocessed data is sent to the inference plugin in a tensor format. 2. The inference plugin sends this data to the machine learning engines: ▫ LiteRT engine uses qtimltflite. ▫ Qualcomm Neural Processing SDK engine uses qtimlsnpe.
Postprocessing	The output tensor is passed down to the postprocessing plugins: ▪ Object detection–qtimlvdetection ▪ Image classification–qtimlvclassification ▪ Image segmentation–qtimlvsegmentation

Table 6-9 ML pipeline processing stages (cont.)

Process	Description
Overlay	The overlay plugin that interprets the machine learning metadata. 1. qtivoverlay draws appropriate overlay on the buffer that allows the inference result to be visualized in real-time. 2. The stream can be rendered on to the display, streamed over network, or encoded and stored to a local storage.
Sink	▪ Waylandsink is used to display the output on a screen. ▪ Filesink is used to save the inferenced video to a local storage.

RELATED INFORMATION

"Run machine learning use cases" on page 212

6.2 Configure Qualcomm GStreamer plugins

The Qualcomm GStreamer plugins control various aspects of the multimedia software. These plugins and the sample applications are available in the gst-plugins-qti-oss folder.

The plugins are categorized as good, bad, and ugly. The Qualcomm GStreamer plugins are categorized as **Good** plugins.

The table lists the attributes that make the plugins good, bad, or ugly:

Table 6-10 Good, bad, ugly GSt plugins

GSt plugin category	Attributes
Good	▪ Good quality code ▪ Well-tested and correct functionality ▪ Preferred license (GNU Lesser General Public License (LGPL))
Bad	▪ Code is close to being good quality ▪ Missing code reviews ▪ Incomplete testing and documentation
Ugly	▪ Good quality code ▪ Well tested and correct functionality ▪ Problematic distribution, which can result in patent issues

For more information, seehttps://gstreamer.freedesktop.org/documentation/additional/splitup.html?gi-language=c.

6.2.1 Configure display, camera, encode and decode plugins

Using these plugins, you can capture, compose, split, transform, decode, encode, and preview the video streams.

RELATED INFORMATION

"Configure audio plugins" on page 389
"Configure ML plugins" on page 397

6.2.1.1 qtiqmmfsrc

The qtiqmmfsrc plugin captures the video frames through Qualcomm Camera Service.

The plugin has the GstQtiQmmfSrc main class that acts as a wrapper on top of the Qualcomm multimedia framework (QMMF) recorder client with separate pads for video and image streams.

- The pads store the creation time parameters (passed as GstCaps during pipeline creation) for the particular stream.
- The GstQtiQmmfSrc takes that information, translates it to the QMMF recorder client parameters, and calls the necessary APIs on each state transition of the element.
- For video and image pads, set the camera device ID through the `camera` property. By default, this ID is 0.

The following are the transition states of the plugin:

- During the transition between the NULL and READY states, the plugin opens and initializes the camera device with the specified ID.
- When transitioning to PAUSED state from READY, the plugin translates the set pad parameters and makes calls to the QMMF service to create the source streams for each pad.
- The session streams are started when transition is done to the PLAYING state.

While the plugin receives a frame in the main class, it creates a GstBuffer and sends it to the relevant pad buffer queue. The pad pushes the buffer to its linked sink pad from the next plugin.

The QMMF service does the following:

- It assigns the buffers. The plugin returns the buffers to the service when they're no longer in use.
- For video, the QMMF service sends the buffers when the plugin state is PLAYING.
- For image pad, a *capture-image* signal must be sent. For each capture-image, the QMMF service sends a single buffer.

Inheritance chain

GObject → GstObject → GstElement → GstQmmfSrc

The following tables provide information on pad templates and element properties of qtiqmmfsrc. For use cases, see Camera.

Pad configuration

Table 6-11 Pad templates for qtiqmmfsrc

Pad Name	Capabilities		
SRC template: 'image_%u' ▪ *Availability:* On request ▪ *Direction:* source	image/jpeg	▪ width: ▪ height: ▪ framerate:	[16, 4096] [16, 4096] [0/1, 30/1]
	video/x-raw	format: width: height: framerate:	{ (string)NV12, (string)NV21 } [16, 4096] [16, 4096] [0/1, 30/1]

Table 6-11 Pad templates for qtiqmmfsrc (cont.)

Pad Name	Capabilities		
	video/x-raw(memory:GBM)	format: width: height: framerate:	{ (string)NV12, (string)NV21 } [16, 4096] [16, 4096] [0/1, 30/1]
	video/x-bayer	format: bpp: width: height: framerate:	{ (string)bggr, (string)rggb, (string)gbrg, (string)grbg, (string)mono } { (string)8, (string)10, (string)12, (string)16 } [16, 4096] [16, 4096] [0/1, 30/1]
SRC template: 'video_%u' ■ *Availability:* On request ■ *Direction:* source	image/jpeg	width: height: framerate:	[16, 4096] [16, 4096] [0/1, 120/1]
	video/x-raw	format: width: height: framerate:	{ (string)NV12, (string)NV16, (string)YUY2 } [16, 4096] [16, 4096] [0/1, 120/1]
	video/x-raw(memory:GBM)	format: width: height: framerate:	{ (string)NV12, (string)YUY2 } [16, 4096] [16, 4096] [0/1, 120/1]
	video/x-bayer	format: bpp: width: height: framerate:	(string)bggr, (string)rggb, (string)gbrg, (string)grbg, (string)mono } { (string)8, (string)10, (string)12, (string)16 } [16, 4096] [16, 4096] [0/1, 120/1]

Table 6-12 Pad properties of qtiqmmfsrc

Pad	Property	Description
'image_%u'	mode	Different image mode. ■ flags: readable, writable, changeable only in NULL or READY state ■ Enum "GstImageMode" Default: 0, "video" □ (0): video - Video snapshot is captured with video settings. Video recording isn't interrupted in this mode □ (1): continuous - Continuously capture images at the set frame rate in the capabilities.
'video_%u'	source-index	Index of the source video pad to which this pad is linked.

Table 6-12 Pad properties of qtiqmmfsrc (cont.)

Pad	Property	Description
		■ flags: readable, writable, changeable only in NULL or READY state ■ Integer. Range: -1 - 2147483647 Default: -1
	framerate	Target framerate in frames per second for display. ■ flags: readable, writable, changeable in NULL, READY, PAUSED, or PLAYING state ■ Double. Range: 0 - 30; default: 30
	crop	Crop rectangle ('<X, Y, WIDTH, HEIGHT>'). Applicable only for JPEG and YUY2 formats. ■ flags: readable, writable, changeable in NULL, READY, PAUSED, or PLAYING state ■ GstValueArray of GValues of type "gint"
	extra-buffers	Number of extra buffers that are allocated. ■ flags: readable, writable, changeable only in NULL or READY state ■ Unsigned Integer. Range: 0 - 4294967295; default: 0
	type	The type of the stream. ■ flags: readable, writable, changeable in NULL, READY, PAUSED, or PLAYING state ■ Enum "GstQmmfSrcStreamType" Default: 0, "video" □ (0): video - The stream is configured with tunings and settings most fitted for directly encoding the buffers. □ (1): preview - The stream is configured with tunings and settings most fitted for directly visualizing the buffers.

Element configuration

Table 6-13 Element properties of qtiqmmfsrc

Property	Description
name	The name of the object. ■ flags: readable, writable ■ String. Default: "qmmfsrc0"
parent	The parent of the object. ■ flags: readable, writable ■ Object of type "GstObject"
camera	The camera device ID to be used by video/image pads. ■ flags: readable, writable ■ Unsigned Integer. Range: 0 - 10 Default: 0
slave	Set the camera as the secondary device. ■ flags: readable, writable ■ Boolean. Default: false
ldc	Lens distortion correction.

Table 6-13 Element properties of qtiqmmfsrc (cont.)

Property	Description
	■ flags: readable, writable ■ Boolean. Default: false
lcac	Lateral chromatic aberration correction. ■ flags: readable, writable ■ Boolean. Default: false
eis	Electronic image stabilization to reduce the effects of camera shake. ■ flags: readable, writable ■ Boolean. Default: false
shdr	Super high dynamic range imaging. ■ flags: readable, writable ■ Boolean. Default: false
adrc	Automatic dynamic range compression. ■ flags: readable, writable ■ Boolean. Default: false
control-mode	Overall mode of auto-exposure, auto-white-balance, auto-focus (also known as 3A) control routines. This is a top-level 3A control switch. When set to OFF, 3A controlled by the camera device is disabled. ■ flags: readable, writable, changeable in NULL, READY, PAUSED or PLAYING state ■ Enum "GstCameraControlMode" Default: 1, "auto" □ (0): off - Full application control of pipeline □ (1): auto - Manual control of capture parameters is disabled □ (2): use-scene-mode - Use a specific scene mode □ (3): off-keep-state - Same as OFF mode, except that this capture isn't used by camera device background 3A to update their statistics.
effect	Effect applied on the camera frames. ■ flags: readable, writable, changeable in NULL, READY, PAUSED or PLAYING state ■ Enum "GstCameraEffectMode" Default: 0, "off" □ (0): off - No color effect is applied. □ (1): mono - A 'monocolor' effect where the image is mapped into a single color. □ (2): negative - A 'photo-negative' effect where the image colors are inverted. □ (3): solarize - A 'solarisation' effect (Sabattier effect) where the image is completely or partially reversed in tone. □ (4): sepia - A 'sepia' effect where the image is mapped into warm gray, red, and brown tones. □ (5): posterize - A 'posterization' effect where the image uses discrete regions of tone rather than a continuous gradient of tones. □ (6): whiteboard - A 'whiteboard' effect where the image is typically displayed as regions of white, with black, or grey details. □ (7): blackboard - A 'blackboard' effect where the image is typically displayed as regions of black, with white, or grey details. □ (8): aqua - An 'aqua' effect where a blue hue is added to the image.
scene	Camera optimizations depending on the scene.

Table 6-13 Element properties of qtiqmmfsrc (cont.)

Property	Description
	■ flags: readable, writable, changeable in NULL, READY, PAUSED or PLAYING state ■ Enum "GstCameraSceneMode" Default: 1, "face-priority" ▫ (0): disabled - Indicates that no scene modes are set ▫ (1): face-priority - Optimized for photos of with priority of people faces ▫ (2): action - Optimized for photos of quickly moving objects. ▫ (3): portrait - Optimized for still photos of people ▫ (4): landscape - Optimized for photos of distant macroscopic objects ▫ (5): night - Optimized for low-light settings ▫ (6): night-portrait - Optimized for still photos of people in low-light settings ▫ (7): theatre - Optimized for dim, indoor settings where flash must remain off ▫ (8): beach - Optimized for bright, outdoor beach settings ▫ (9): snow - Optimized for bright, outdoor settings containing snow ▫ (10): sunset - Optimized for scenes of the setting sun ▫ (11): steady-photo - Optimized to avoid blurry photos due to small amounts of device motion (for example: due to handshake) ▫ (12): fireworks - Optimized for nighttime photos of fireworks ▫ (13): sports - Optimized for photos of fast moving people ▫ (14): party - Optimized for dim, indoor settings with several moving people ▫ (15): candlelight - Optimized for dim settings where the main light source is a candle ▫ (16): hdr - Turn on a device-specific high dynamic range (HDR) mode
antibanding	Camera antibanding routine for the current illumination condition. ■ flags: readable, writable, changeable in NULL, READY, PAUSED or PLAYING state ■ Enum "GstAntibandingMode" Default: 3, "auto" ▫ (0): off - The camera device doesn't adjust exposure duration to avoid banding problems ▫ (1): 50 Hz - The camera device adjusts exposure duration to avoid banding problems with 50 Hz illumination sources ▫ (2): 60 Hz - The camera device adjusts exposure duration to avoid banding problems with 60 Hz illumination sources ▫ (3): auto - The camera device automatically adapts its antibanding routine to the current illumination condition
sharpness	Image sharpness strength. ■ flags: readable, writable, changeable in NULL, READY, PAUSED or PLAYING state ■ Integer. Range: 0 - 6; default: 2
contrast	Image contrast strength. ■ flags: readable, writable, changeable in NULL, READY, PAUSED or PLAYING state ■ Integer. Range: 1 - 10; default: 5
saturation	Image saturation strength. ■ flags: readable, writable, changeable in NULL, READY, PAUSED or PLAYING state ■ Integer. Range: 0 - 10 Default: 5

Table 6-13 Element properties of qtiqmmfsrc (cont.)

Property	Description
iso-mode	ISO exposure mode. ■ flags: readable, writable, changeable in NULL, READY, PAUSED or PLAYING state ■ Enum "GstCameraISOMode" Default: 0, "auto" ▫ (0): auto - The chosen ISO exposure mode depending on the scene ▫ (1): deblur - The ISO exposure sensitivity set to prioritize motion deblur ▫ (2): 100 - The ISO exposure sensitivity set to prioritize level 100 ▫ (3): 200 - The ISO exposure sensitivity set to prioritize level 200 ▫ (4): 400 - The ISO exposure sensitivity set to prioritize level 400 ▫ (5): 800 - The ISO exposure sensitivity set to prioritize level 800 ▫ (6): 1600 - The ISO exposure sensitivity set to prioritize level 1600 ▫ (7): 3200 - The ISO exposure sensitivity set to prioritize level 3200 ▫ (8): manual - The ISO exposure value provided by manual-iso-value is used
manual-iso-value	Manual exposure ISO value. Used when the ISO mode is set to 'manual'. ■ flags: readable, writable, changeable in NULL, READY, PAUSED or PLAYING state ■ Integer. Range: 100 - 3200 Default: 800
exposure-mode	The required mode for the camera exposure routine. ■ flags: readable, writable, changeable in NULL, READY, PAUSED or PLAYING state ■ Enum "GstCameraExposureMode" Default: 1, "auto" ▫ (0): off - The auto exposure routine is disabled. Manual exposure time is used to set the 'exposure-time' property ▫ (1): auto - The auto exposure routine is active
exposure-lock	Locks current camera exposure routine values from changing. ■ flags: readable, writable, changeable in NULL, READY, PAUSED or PLAYING state ■ Boolean. Default: false
exposure-metering	The required mode for the camera exposure metering routine. ■ flags: readable, writable, changeable in NULL, READY, PAUSED or PLAYING state ■ Enum "GstCameraExposureMetering" Default: 0, "average" ▫ (0): average - The camera exposure metering is calculated as average from the whole frame. ▫ (1): center-weighted - The camera exposure metering is calculated from the center region of the frame. ▫ (2): spot - The camera exposure metering is calculated from a chosen spot. ▫ (6): custom - The camera exposure metering is calculated from a custom metering table.
exposure-compensation	Adjust (Compensate) camera images target brightness. The adjustment is measured as a count of steps. ■ flags: readable, writable, changeable in NULL, READY, PAUSED or PLAYING state ■ Integer. Range: -12 - 12 Default: 0
manual-exposure-time	Manual exposure time in nanoseconds. Used when the Exposure mode is set to 'off'. ■ flags: readable, writable, changeable in NULL, READY, PAUSED or PLAYING state ■ Integer64. Range: 0 - 9223372036854775807 Default: 33333333

Table 6-13 Element properties of qtiqmmfsrc (cont.)

Property	Description
custom-exposure-table	A GstStructure describing custom exposure table. ■ flags: readable, writable, changeable in NULL, READY, PAUSED or PLAYING state ■ String. Default: "org.codeaurora.qcamera3.exposuretable;"
white-balance-mode	The mode for the camera's white balance routine. ■ flags: readable, writable, changeable in NULL, READY, PAUSED or PLAYING state ■ Enum "GstCameraWiteBalanceMode" Default: 3, "auto" □ (0): off - Both auto and manual white balance routines are disabled □ (1): manual-cc-temp - The auto-white balance routine is inactive and manual color correction temperature is used, which is set through the 'manual-wb-settings' property □ (2): manual-rgb-gains - The auto-white balance routine is inactive and manual R/G/B gains are used, which are set through the 'manual-wb-settings' property □ (3): auto - The auto-white balance routine is active □ (4): shade - The camera device uses shade light as the assumed scene illumination for white balance correction □ (5): incandescent - The camera device uses incandescent light as the assumed scene illumination for white balance correction □ (6): fluorescent - The camera device uses fluorescent light as the assumed scene illumination for white balance correction □ (7): warm-fluorescent - The camera device uses warm fluorescent light as the assumed scene illumination for white balance correction □ (8): daylight - The camera device uses daylight light as the assumed scene illumination for white balance correction □ (9): cloudy-daylight - The camera device uses cloudy daylight light as the assumed scene illumination for white balance correction □ (10): twilight - The camera device uses twilight light as the assumed scene illumination for white balance correction
white-balance-lock	Locks current White Balance values from changing. Affects only non-manual white balance modes. ■ flags: readable, writable, changeable in NULL, READY, PAUSED or PLAYING state ■ Boolean. Default: false
manual-wb-settings	Manual White Balance settings such as color correction temperature and R/G/B gains. Used in manual white balance modes. ■ flags: readable, writable, changeable in NULL, READY, PAUSED or PLAYING state ■ String. Default: "org.codeaurora.qcamera3.manualWB;"
focus-mode	Indicates whether auto-focus is enabled along with its mode. ■ flags: readable, writable, changeable in NULL, READY, PAUSED or PLAYING state ■ Enum "GstCameraFocusMode" Default: 0, "off" □ (0): off - The auto focus routine is disabled □ (1): auto - The auto focus routine is active □ (2): macro - In this mode, the auto focus algorithm is optimized for focusing on objects close to the camera □ (3): continuous - In this mode, the AF algorithm modifies the lens position continually to try to provide a constantly-in-focus image stream □ (4): edof - The camera device produces images with an extended depth of field automatically; no special focusing operations must be done before taking a picture
noise-reduction	Noise reduction filter mode.

Table 6-13 Element properties of qtiqmmfsrc (cont.)

Property	Description
	■ flags: readable, writable, changeable in NULL, READY, PAUSED or PLAYING state ■ Enum "GstCameraNoiseReduction" Default: 1, "fast" ▫ (0): off - No noise reduction filter is applied. ▫ (1): fast - TNR (Temporal Noise Reduction) Fast Mode ▫ (2): hq - TNR (Temporal Noise Reduction) High Quality Mode
noise-reduction-tuning	A GstStructure describing the noise reduction tuning. ■ flags: readable, writable, changeable in NULL, READY, PAUSED or PLAYING state ■ String. Default: "org.quic.camera.anr_tuning;"
zoom	Camera zoom rectangle ('<X, Y, WIDTH, HEIGHT >') in sensor active pixel array coordinates. ■ flags: readable, writable, changeable in NULL, READY, PAUSED or PLAYING state ■ GstValueArray of GValues of type "gint"
defog-table	A GstStructure describing defog table. ■ flags: readable, writable, changeable in NULL, READY, PAUSED or PLAYING state ■ String. Default: "org.quic.camera.defog;"
ltm-data	A GstStructure describing local tone mapping data. ■ flags: readable, writable, changeable in NULL, READY, PAUSED or PLAYING state ■ String. Default: "org.quic.camera.ltmDynamicContrast;"
infrared-mode	Infrared mode. ■ flags: readable, writable, changeable in NULL, READY, PAUSED or PLAYING state ■ Enum "GstCameraIRMode" Default: 0, "off" ▫ (0): off - The infrared LED is OFF and cut filter is applied, that is, infrared light is blocked. ▫ (1): on - The infrared LED is ON and cut filter is removed, that is, infrared light is allowed. ▫ (2): auto - The infrared LED and cut filter are turned ON or OFF depending on the conditions. ▫ (3): cut-filter-only - The infrared LED is turned OFF and cut filter is applied, that is, IR light is blocked. ▫ (4): cut-filter-disable - Infrared cut filter is removed allowing IR light to pass. This mode is used for transitioning from 'cut-filter-only' mode, that is, disabling only the cut filter.
active-sensor-size	The active pixel array of the camera sensor ('<X, Y, WIDTH, HEIGHT >') and it's filled only when the plugin is in READY or above state. ■ flags: readable, changeable in NULL, READY, PAUSED or PLAYING state ■ GstValueArray of GValues of type "gint"
sensor-mode	Force set Sensor Mode index (0-15). -1 for Auto selection. ■ flags: readable, writable ■ Integer. Range: -1 - 15 Default: -1
capture-metadata	Expose camera metadata object that's used for camera control. If the get property is used, caller must release metadata object.

Table 6-13 Element properties of qtiqmmfsrc (cont.)

Property	Description
	▪ flags: readable, writable, changeable in NULL, READY, PAUSED or PLAYING state ▪ Pointer
camera-characteristics	Returns supported values for camera parameters as camera metadata object. The caller should take ownership and release the metadata object. ▪ flags: readable, changeable in NULL, READY, PAUSED or PLAYING state ▪ Pointer

Table 6-14 Element signals for qtiqmmfsrc

Signal	Function
result-metadata	`void user_function (GstElement* object, gpointer arg0, gpointer user_data);`
urgent-metadata	`void user_function (GstElement* object, gpointer arg0, gpointer user_data);`

6.2.1.2 qtivcomposer

The qtivcomposer plugin uses the GPU hardware to merge/mix several input video streams into a single output stream.

The pad properties determine the composition of the input buffers (for each stream) in the output buffer.

The table lists a few common pad properties of qtivcomposer. For more information, seeTable 6-16.

Property	Usage
Position	Sets the X and Y axes placement of each input frame within the output.
Crop	Sets the optional crop for the source.
Dimensions	Upscales/downscales the frames.
Rotate	Rotates the frames at right angles
▪ Flip-horizontal ▪ Flip-vertical	Flips the frames horizontally and vertically.
Alpha	Adds the alpha blending value to the frames.
Z order	Changes the order of the frames. By default, the frames are organized in the order in which the sink pads are created. For example, sink_0, and sink_1.

The figure shows the GStreamer pipeline using qtivcomposer that's working with several video sources.

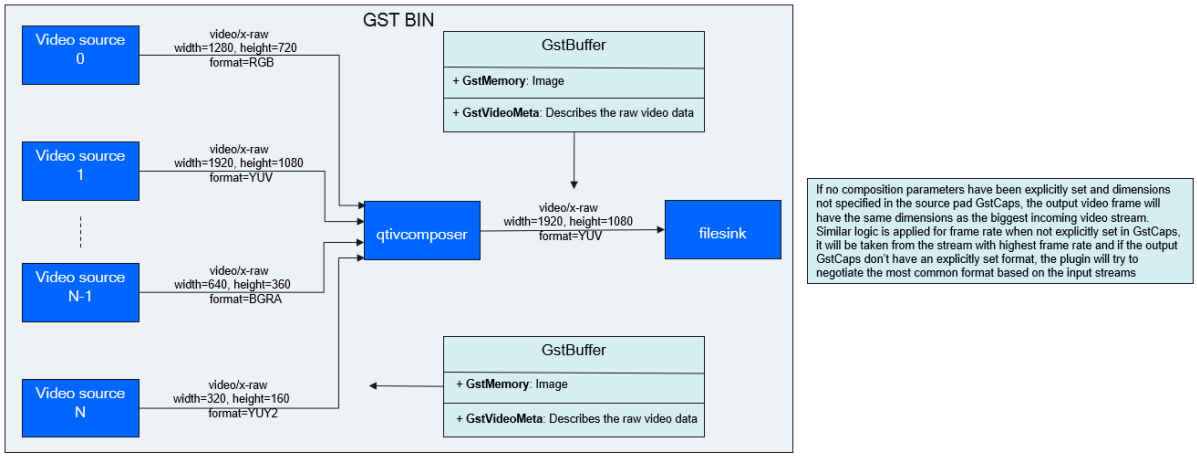


Figure 6-11 Pipeline with qtivcomposer

The video compositions include, but aren't limited to the following:

- Picture-in-picture
- Side-by-side
- Alpha blending

If no composition parameters and dimensions are specified in the GstCaps source pad, the output video frames will have the same dimensions as the biggest incoming video stream.

Similarly, if the frame rate isn't set in GstCaps, it's taken from the stream with the highest frame rate. And if output GstCaps doesn't have an explicitly set format, the plugin negotiates the most common format based on the input streams.

The plugin uses the Qualcomm IB2C library or Qualcomm Computer Vision SDK for all the transformation operations.

- The Qualcomm IB2C library is wrapped in the custom GstC2dVideoConverter or GstGlesVideoConverter abstraction layer.
- It contains APIs to create, configure, and process the incoming and outgoing buffers.
- The recommended plugin is GstGLESVideoConverter.

A custom buffer pool class called GstImageBufferPool allocates the output buffers. GstImageBufferPool allocates either GBM or ION buffers:

- The GBM allocation is done through the Qualcomm libgbm.
- The ION allocation is done through IOCTL commands to the kernel.

Inheritance chain

GObject → GstObject → GstElement → GstAggregator → GstVideoComposer

The following tables provide information on pad templates and element properties of qtivcomposer. For use cases, see Video composition use cases.

Pad configuration

Table 6-15 Pad templates for qtivcomposer

Pad Name	Capabilities		
SINK template: 'sink_%u' ■ *Availability:* On request ■ *Direction:* sink	video/x-raw	format:	{ (string)NV12, (string)NV21, (string)UYVY, (string)YUY2, (string)RGBA, (string)BGRA, (string)ARGB, (string)ABGR, (string)RGBx, (string)BGRx, (string)xRGB, (string)xBGR, (string)RGB, (string)BGR, (string)GRAY8 }
		width:	[1, 32767]
		height:	[1, 32767]
		framerate:	[0/1, 255/1]
	video/x-raw(memory:GBM)	format:	{ (string)NV12, (string)NV21, (string)UYVY, (string)YUY2, (string)RGBA, (string)BGRA, (string)ARGB, (string)ABGR, (string)RGBx, (string)BGRx, (string)xRGB, (string)xBGR, (string)RGB, (string)BGR, (string)GRAY8 }
		width:	[1, 32767]
		height:	[1, 32767]
		framerate:	[0/1, 255/1]
SRC template: 'src' ■ *Availability:* Always ■ *Direction:* source	video/x-raw	format:	{ (string)NV12, (string)NV21, (string)UYVY, (string)YUY2, (string)RGBA, (string)BGRA, (string)ARGB, (string)ABGR, (string)RGBx, (string)BGRx, (string)xRGB, (string)xBGR, (string)RGB, (string)BGR, (string)GRAY8 }
		width:	[1, 32767]
		height:	[1, 32767]
		framerate:	[0/1, 255/1]
	video/x-raw(memory:GBM)	format:	{ (string)NV12, (string)NV21, (string)UYVY, (string)YUY2, (string)RGBA, (string)BGRA, (string)ARGB, (string)ABGR, (string)RGBx, (string)BGRx, (string)xRGB, (string)xBGR, (string)RGB, (string)BGR, (string)GRAY8 }
		width:	[1, 32767]
		height:	[1, 32767]
		framerate:	[0/1, 255/1]

Table 6-16 Pad properties of qtivcomposer

Pad	Property	Description
'sink_%u'	zorder	Z axis order, the default is the order of creation. ■ flags: readable, writable, changeable in NULL, READY, PAUSED or PLAYING state, 0x40000000 ■ Integer. Range: -1 to 2147483647 Default: -1
	crop	The crop rectangle ('<X, Y, WIDTH, HEIGHT >'). ■ flags: readable, writable, changeable in NULL, READY, PAUSED or PLAYING state, 0x40000000 ■ GstValueArray of GValues of type "gint"
	position	The X and Y coordinates of the destination rectangle upper left corner ('<X, Y>'). ■ flags: readable, writable, changeable in NULL, READY, PAUSED or PLAYING state, 0x40000000 ■ GstValueArray of GValues of type "gint"
	dimensions	The destination rectangle width and height, if left as '0' they're the same as input dimensions ('<WIDTH, HEIGHT>') ■ flags: readable, writable, changeable in NULL, READY, PAUSED, or PLAYING state, 0x40000000 ■ GstValueArray of GValues of type "gint"
	alpha	Alpha channel value. ■ flags: readable, writable, changeable in NULL, READY, PAUSED, or PLAYING state, 0x40000000 ■ Double. Range: 0.0 - 1.0 Default: 1.0
	flip-horizontal	Flip the video image horizontally. ■ flags: readable, writable, changeable in NULL, READY, PAUSED, or PLAYING state, 0x40000000 ■ Boolean. Default: false
	flip-vertical	Flip the video image vertically. ■ flags: readable, writable, changeable in NULL, READY, PAUSED, or PLAYING state, 0x40000000 ■ Boolean. Default: false
	rotate	Rotate the video image. ■ flags: readable, writable, changeable in NULL, READY, PAUSED, or PLAYING state, 0x40000000 ■ Enum "GstVideoComposerRotate" Default: 0, "none" □ (0): none - No rotation □ (1): 90CW - Rotate 90 degrees clockwise □ (2): 90CCW - Rotate 90 degrees counter-clockwise □ (3): 180 - Rotate 180 degrees
	emit-signals	Send signals to signal data consumption. ■ flags: readable, writable ■ Boolean. Default: false

Element configuration

Table 6-17 Element properties of qtivcomposer

Property	Description
name	The name of the object. ■ flags: readable, writable ■ String. Default: "videocomposer0"
parent	The parent of the object. ■ flags: readable, writable ■ Object of type "GstObject"
engine	Engine backend used for the conversion operations. ■ flags: readable, writable ■ Enum "GstVideoConverterBackend" Default: 1, "gles" □ (1): gles - Use OpenGLES based video converter □ (2): fcv - Use FastCV based video converter **NOTE** The Qualcomm Computer Vision SDK (fcv) engine is currently not supported.
latency	Additional latency in live mode allows upstream to take longer to produce buffers for the current position (in nanoseconds). ■ flags: readable, writable ■ Unsigned Integer64. ■ Range: 0 - 18446744073709551615 ■ Default: 0
start-time-selection	Decides which start time is output. ■ flags: readable, writable ■ Enum "GstAggregatorStartTimeSelection" Default: 0, "zero" □ (0): zero - Start at 0 running time (default) □ (1): first - Start at the first observed input running time □ (2): set - Set start time with start-time property
start-time	Start time to use if start-time-selection=set. ■ flags: readable, writable ■ Unsigned Integer64. Range: 0 - 18446744073709551615 Default: 18446744073709551615
background	Background color. ■ flags: readable, writable, changeable in NULL, READY, PAUSED, or PLAYING state ■ Unsigned Integer Range: 0 - 4294967295 ■ Default: 4286611584

6.2.1.3 Waylandsink

The waylandsink plugin uses the GstVideoSink GStreamer class. It's based on the Wayland Weston compositor, which creates its own window and renders the incoming video frames in the window.

It maps the Weston client APIs and states to the appropriate GStreamer APIs and states. Qualcomm has extended the waylandsink implementation and added support for window positioning, a proprietary GBM-based buffer backend for zero-copy and XDG shell backend.

Inheritance chain

GObject → GstObject → GstElement → GstBaseSink → GstVideoSink → GstWaylandSink

The following tables provide information on pad templates and element properties of waylandsink. For use cases, see Video playback use cases.

Pad configuration

Table 6-18 Pad templates for Waylandsink

Pad Name	Capabilities		
SINK template: 'sink' Availability:Always Direction: sink	video/x-raw	format:	{ (string)BGRx, (string)BGRA, (string)RGBx, (string)xBGR, (string)xRGB, (string)RGBA, (string)ABGR, (string)ARGB, (string)RGB, (string)BGR, (string)RGB16, (string)BGR16, (string)YUY2, (string)YVYU, (string)UYVY, (string)AYUV, (string)NV12, (string)NV21, (string)NV16, (string)YUV9, (string)YVU9, (string)Y41B, (string)I420, (string)YV12, (string)Y42B, (string)v308 }
		width:	[1, 2147483647]
		height:	[1, 2147483647]
		framerate:	[0/1, 2147483647/1]
	video/x-raw(memory:DMABuf)	format:	{ (string)BGRx, (string)BGRA, (string)RGBx, (string)xBGR, (string)xRGB, (string)RGBA, (string)ABGR, (string)ARGB, (string)RGB, (string)BGR, (string)RGB16, (string)BGR16, (string)YUY2, (string)YVYU, (string)UYVY, (string)AYUV, (string)NV12, (string)NV21, (string)NV16, (string)YUV9, (string)YVU9, (string)Y41B, (string)I420, (string)YV12, (string)Y42B, (string)v308 }
		width:	[1, 2147483647]
		height:	[1, 2147483647]
		framerate:	[0/1, 2147483647/1]

Table 6-18 Pad templates for Waylandsink (cont.)

Pad Name	Capabilities		
	video/x-raw	format:	{ (string)BGRx, (string)BGRA, (string)RGBx, (string)xBGR, (string)xRGB, (string)RGBA, (string)ABGR, (string)ARGB, (string)RGB, (string)BGR, (string)RGB16, (string)BGR16, (string)YUY2, (string)YVYU, (string)UYVY, (string)AYUV, (string)NV12, (string)NV21, (string)NV16, (string)YUV9, (string)YVU9, (string)Y41B, (string)I420, (string)YV12, (string)Y42B, (string)v308 }
		width:	[1, 2147483647]
		height:	[1, 2147483647]
		framerate:	[0/1, 2147483647/1]

Element configuration

NOTE For the current release, **x**, **y**, **width**, and **height** properties aren't supported.

Table 6-19 Element properties of Waylandsink

Property	Description
name	The name of the object. ▪ flags: readable, writable ▪ String. Default: "waylandsink0"
parent	The parent of the object. ▪ flags: readable, writable ▪ Object of type "GstObject"
sync	Sync on the clock. ▪ flags: readable, writable ▪ Boolean. Default: true
max-lateness	Maximum number of nanoseconds that a buffer can be late before it's dropped (-1 unlimited). ▪ flags: readable, writable ▪ Integer64. Range: -1 to 9223372036854775807, default: 20000000
qos	Generate quality-of-service events upstream. ▪ flags: readable, writable ▪ Boolean. Default: true
async	Go asynchronously to PAUSED. ▪ flags: readable, writable ▪ Boolean. Default: true
ts-offset	Timestamp offset in nanoseconds. ▪ flags: readable, writable ▪ Integer64. Range: -9223372036854775808 to 9223372036854775807, default: 0

Table 6-19 Element properties of Waylandsink (cont.)

Property	Description
enable-last-sample	Enable the last-sample property. ▪ flags: readable, writable ▪ Boolean. Default: true
last-sample	The last sample that was received in the sink. ▪ flags: readable ▪ Boxed pointer of type "GstSample"
blocksize	Size in bytes to pull per buffer (0 = default). ▪ flags: readable, writable ▪ Unsigned Integer. Range: 0 - 4294967295 Default: 4096
render-delay	More render delay of the sink in nanoseconds. ▪ flags: readable, writable ▪ Unsigned Integer64. Range: 0 - 18446744073709551615 Default: 0
throttle-time	The time to keep between rendered buffers (0 = disabled). ▪ flags: readable, writable ▪ Unsigned Integer64. Range: 0 - 18446744073709551615 Default: 0
max-bitrate	The maximum bits per second to render (0 = disabled). ▪ flags: readable, writable ▪ Unsigned Integer64. Range: 0 - 18446744073709551615 Default: 0
show-preroll-frame	Render or not render the video frames during preroll. ▪ flags: readable, writable ▪ Boolean. Default: true
display	The Wayland display name that can be used to make a connection, if it isn't supplied using the GstContext. ▪ flags: readable, writable ▪ String. Default: null
xdg-shell	Use the XDG shell protocol for the display. ▪ flags: readable, writable ▪ Boolean. Default: true
fullscreen	Determine if the surface should be full screen. ▪ flags: readable, writable, changeable in NULL, READY, PAUSED, or PLAYING state ▪ Boolean. Default: false

6.2.1.4 qtivsplit

The qtivsplit plugin uses the GPU hardware to split a single input video stream into multiple output streams. The number of streams is a user-defined variable and is equal to the number of source pads.

The mode property for each source pad determines the method for splitting the incoming stream.

Table 6-20 Operational mode properties of qtivsplit

Operational mode	Description
none	▪ The incoming video frames on the sink pad are duplicated on each of the source pads. ▪ More color conversion and upscale/downscale are applied based on the negotiated GstCaps on that pad.
single-roi-meta	▪ The input video buffers are checked for GstVideoRegionOfInterestMeta. ▪ Ensure that when the source pads are created, their number is equal to the maximum expected count of GstVideoRegionOfInterestMeta inside a single GstBuffer. ▪ If their source pads are fewer, the ROI meta that doesn't have a corresponding pad is ignored.
batch-roi-meta	Operates the GstVideoRegionOfInterestMeta entries attached to the buffer but instead of sending one such entry, it sends all the meta entries on the pads in that mode as separate buffers.

Each meta entry in the GstBuffer is sent to its corresponding source pad based on the ID and crop, upscale/downscale, and color conversion is performed to match the negotiated GstCaps. The source pads with no corresponding ROI meta produce GAP buffers.

The plugin uses the Qualcomm IB2C library or Qualcomm Computer Vision SDK for all the transformation operations.

- The Qualcomm IB2C library is wrapped in the custom GstC2dVideoConverter or GstGlesVideoConverter abstraction layer.
- It includes APIs to create, configure, and process the incoming and outgoing buffers.

Buffer management

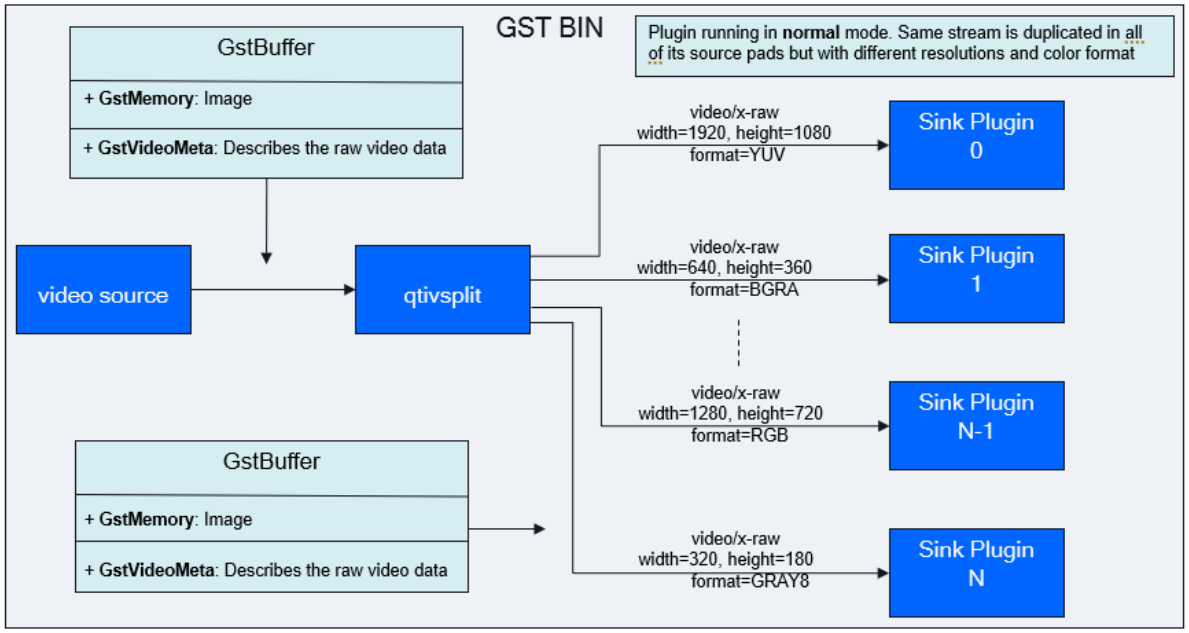


Figure 6-12 GStreamer pipeline with qtivsplit

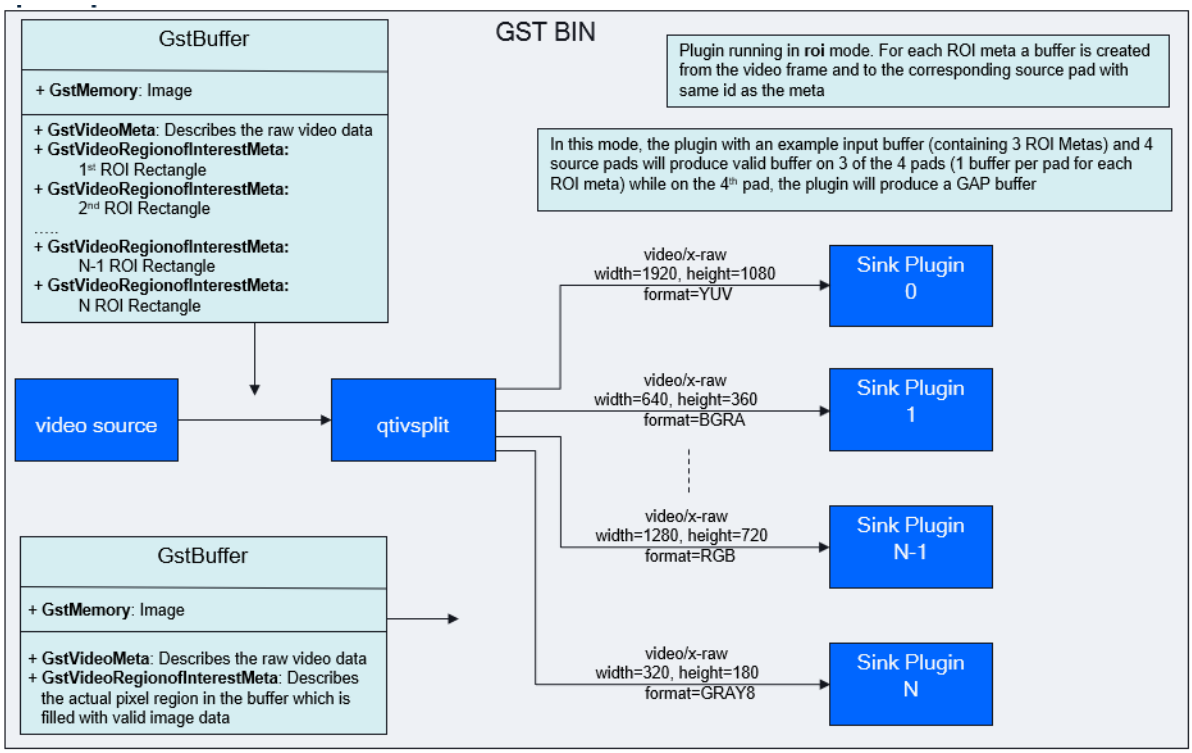


Figure 6-13 qtivsplit buffer management

The GstImageBufferPool custom buffer pool class allocates either GBM or ION output buffers depending on the negotiated capabilities between GstImageBufferPool and downstream plugins.

- The GBM allocation is done using Qualcomm libgbm.
- The ION allocation is done using the IOCTL commands to the kernel.

Inheritance chain

GObject → GstObject → GstElement → GstVideoSplit

The following tables provide information on pad templates and element properties of qtivsplit. For use cases, see the following:

- For use case on splitting video into several streams, see Video split use cases.
- For use case on dynamic cropping, see Daisy chain detection and classification.

Pad configuration

Table 6-21 Pad templates for qtivsplit

Pad Name	Capabilities		
SINK template: 'sink' ▪ *Availability:* Always ▪ *Direction:* sink	video/x-raw	format:	{ (string)NV12, (string)NV21, (string)UYVY, (string)YUY2, (string)RGBA, (string)BGRA, (string)ARGB, (string)ABGR, (string)RGBx, (string)BGRx, (string)xRGB, (string)xBGR, (string)RGB, (string)BGR, (string)GRAY8 }
		width:	[1, 32767]
		height:	[1, 32767]
		framerate:	[0/1, 255/1]
	video/x-raw(memory:GBM)	format:	{ (string)NV12, (string)NV21, (string)UYVY, (string)YUY2, (string)RGBA, (string)BGRA, (string)ARGB, (string)ABGR, (string)RGBx, (string)BGRx, (string)xRGB, (string)xBGR, (string)RGB, (string)BGR, (string)GRAY8 }
		width:	[1, 32767]
		height:	[1, 32767]
		framerate:	[0/1, 255/1]
SRC template: 'src_%u' ▪ *Availability:* On request ▪ *Direction:* source	video/x-raw	format:	{ (string)NV12, (string)NV21, (string)UYVY, (string)YUY2, (string)RGBA, (string)BGRA, (string)ARGB, (string)ABGR, (string)RGBx, (string)BGRx, (string)xRGB, (string)xBGR, (string)RGB, (string)BGR, (string)GRAY8 }
		width:	[1, 32767]

Table 6-21 Pad templates for qtivsplit (cont.)

Pad Name	Capabilities		
		height:	[1, 32767]
		framerate:	[0/1, 255/1]
	video/x-raw(memory:GBM)	format:	{ (string)NV12, (string)NV21, (string)UYVY, (string)YUY2, (string)RGBA, (string)BGRA, (string)ARGB, (string)ABGR, (string)RGBx, (string)BGRx, (string)xRGB, (string)xBGR, (string)RGB, (string)BGR, (string)GRAY8 }
		width:	[1, 32767]
		height:	[1, 32767]
		framerate:	[0/1, 255/1]

Table 6-22 Pad properties for qtivsplit

Pad	Property	Description
'src_%u'	mode	Operational mode ▪ flags: readable, writable, changeable only in NULL or READY state ▪ Enum "GstVideoSplitMode" Default: 0, "none" ▫ (0): none - Incoming buffer is rescaled and color converted to match the negotiated pad caps. If the input and output caps match, then the input buffer is moved to the output and its reference count increased. ▫ (1): force-transform - Incoming buffer is rescaled and color converted to match the negotiated pad caps. A new buffer is created even if the negotiated input and output caps match. ▫ (2): single-roi-meta - Incoming buffer is checked for ROI meta. If there is a meta entry that corresponds to this pad, the crop, rescale and color conversion operations are performed on the input buffer. The transformed buffer is sent to the next plugin. A pad with no corresponding ROI meta creates a GAP buffer. ▫ (3): batch-roi-meta - Incoming buffer is checked for ROI meta. For each meta entry a crop, rescaling and color conversion are done on the input buffer. This, for each ROI meta entry, a buffer is produced and sent to the next plugin downstream. In case ROI meta isn't present, the pad creates a GAP buffer.

Element configuration

Table 6-23 Element properties of qtivsplit

Property	Description
name	The name of the object. ▪ flags: readable, writable ▪ String. Default: "videosplit0"
parent	The parent of the object. ▪ flags: readable, writable ▪ Object of type "GstObject"
engine	Engine backend used for the conversion operations.

Table 6-23 Element properties of qtivsplit (cont.)

Property	Description
	▪ flags: readable, writable ▪ Enum "GstVideoConverterBackend" Default: 1, "gles" ▫ (1): gles - Uses OpenGLES based video converter (2): ▫ fcv - Uses Qualcomm Computer Vision SDK based video converter **NOTE** The Qualcomm Computer Vision SDK (fcv) engine is currently not supported.

6.2.1.5 qtivtransform

The qtivtransform plugin uses GPU hardware to upscale/downscale, flip, rotate, crop, and color covert the incoming YUV or RGB video frames determined by the element properties.

The table lists some of the common properties of qtivtransform. For more information, see Table 6-25.

Property	Usage
Destination	Sets the X and Y axes placement with upscale/downscale dimensions for each input frame within the output.
Crop	Sets the optional crop for the source.
Rotate	Rotates the frames at right angles.
▪ Flip-horizontal ▪ Flip-vertical	Flips the frames horizontally and vertically.

If the GstCaps source pad transformation property isn't specified, the plugin negotiates the same capabilities as provided in the input. In this case, the plugin operates in Passthrough mode until a parameter changes.

The plugin uses the Qualcomm IB2C library exposed by the Qualcomm® Adreno™ GPU for all the transformation operations. This library is wrapped inside the custom GstC2dVideoConverter abstraction layer with APIs to create, configure, and process the incoming and outgoing buffers.

The GstImageBufferPool custom buffer pool class can allocate either GBM or ION output buffers depending on the negotiated capabilities between GstImageBufferPool and the downstream plugins.

- The GBM allocation is done through the Qualcomm libgbm.
- The ION allocation is done through IOCTL commands to the kernel.

Inheritance chain

GObject → GstObject → GstElement → GstBaseTransform → GstVideoTransform

The following tables provide information on pad templates and element properties of qtivtransform. For use cases, see Transform and Transcode use cases.

Pad configuration

Table 6-24 Pad templates for qtivtransform

Pad Name	Capabilities		
SINK template: 'sink' ▪ *Availability:* Always ▪ *Direction:* sink	video/x-raw	format:	{ (string)NV12, (string)NV21, (string)YUY2, (string)P010_10LE, (string)NV12_10LE32, (string)RGBA, (string)BGRA, (string)ARGB, (string)ABGR, (string)RGBx, (string)BGRx, (string)xRGB, (string)xBGR, (string)RGB, (string)BGR, (string)GRAY8 }
		width:	[1, 32767]
		height:	[1, 32767]
		framerate:	[0/1, 255/1]
	video/x-raw(memory:GBM)	format:	{ (string)NV12, (string)NV21, (string)YUY2, (string)P010_10LE, (string)NV12_10LE32, (string)RGBA, (string)BGRA, (string)ARGB, (string)ABGR, (string)RGBx, (string)BGRx, (string)xRGB, (string)xBGR, (string)RGB, (string)BGR, (string)GRAY8 }
		width:	[1, 32767]
		height:	[1, 32767]
		framerate:	[0/1, 255/1]
SRC template: 'src' ▪ *Availability:* Always ▪ *Direction:* source	video/x-raw	format:	{ (string)NV12, (string)NV21, (string)YUY2, (string)P010_10LE, (string)RGBA, (string)BGRA, (string)ARGB, (string)ABGR, (string)RGBx, (string)BGRx, (string)xRGB, (string)xBGR, (string)RGB, (string)BGR, (string)GRAY8 }
		width:	[1, 32767]
		height:	[1, 32767]
		framerate:	[0/1, 255/1]
	video/x-raw(memory:GBM)	format:	{ (string)NV12, (string)NV21, (string)YUY2, (string)P010_10LE, (string)RGBA, (string)BGRA, (string)ARGB, (string)ABGR, (string)RGBx, (string)BGRx, (string)xRGB, (string)xBGR, (string)RGB, (string)BGR, (string)GRAY8 }
		width:	[1, 32767]
		height:	[1, 32767]
		framerate:	[0/1, 255/1]

Element configuration

Table 6-25 Elements properties of qtivtransform

Property	Description
name	The name of the object.

Table 6-25 Elements properties of qtivtransform (cont.)

Property	Description
	■ flags: readable, writable ■ String. Default: "videotransform0"
parent	The parent of the object. ■ flags: readable, writable ■ Object of type "GstObject"
engine	Engine backend used for the conversion operations. ■ flags: readable, writable ■ Enum "GstVideoConverterBackend" Default: 1, "gles" ▫ (1): gles - Use OpenGLES based video converter ▫ (2): fcv - Use Qualcomm Computer Vision SDK based video converter **NOTE** The Qualcomm Computer Vision SDK (fcv) engine is currently not supported.
flip-horizontal	Flip the video image horizontally. ■ flags: readable, writable, changeable in NULL, READY, PAUSED, or PLAYING state ■ Boolean. Default: false
flip-vertical	Flip the video image vertically. ■ flags: readable, writable, changeable in NULL, READY, PAUSED, or PLAYING state ■ Boolean. Default: false
rotate	Rotate video image. ■ flags: readable, writable, changeable in NULL, READY, PAUSED, or PLAYING state ■ Enum "GstVideoTransformRotate" Default: 0, "none" ▫ (0): none - No rotation ▫ (1): 90CW - Rotate 90 degrees clockwise ▫ (2): 90CCW - Rotate 90 degrees counter-clockwise ▫ (3): 180 - Rotate 180 degrees
crop	The crop rectangle inside the input ('<X, Y, WIDTH, HEIGHT >'). **NOTE** This property can't be time synchronized. ■ flags: readable, writable, changeable in NULL, READY, PAUSED or PLAYING state ■ GstValueArray of GValues of type "gint"
destination	Destination rectangle inside the output ('<X, Y, WIDTH, HEIGHT >'). ■ flags: readable, writable, changeable in NULL, READY, PAUSED or PLAYING state ■ GstValueArray of GValues of type "gint"
background	Background color. ■ flags: readable, writable, changeable in NULL, READY, PAUSED or PLAYING state ■ Unsigned Integer. Range: 0 - 4294967295 Default: 4286611584

6.2.1.6 qtisocketsink

The qtisocketsink plugin uses UNIX sockets to transfer file descriptor (FD) backed GstBuffers to other processes with qtisocketsrc as the entry point.

The qtisocketsink plugin requires a UNIX domain socket file (which should have the extension of .sock) to be passed as the *socket* property to transfer the FD.

The buffers passed from the other process to this plugin are tracked with a reference counter, which is incremented or decremented accordingly when the buffers are sent or returned through the socket.

The following are some of the configurations that can be performed using qtisocketsink:

Table 6-26 Configuration properties of qtisocketsink

Property	Configuration
async	Set to false for the sockets to transition asynchronously to the PAUSED state.
max-lateness	Adjust the maximum number of nanoseconds that a buffer can be late before it's dropped.
max-bitrate	Set the maximum bits per second to render.

Inheritance chain

GObject → GstObject → GstElement → GstBaseSink → GstFdSocketSink

The following tables provide information on pad templates and element properties of qtisocketsink. For use cases, see Single camera stream with socket.

Pad configuration

Table 6-27 Pad templates for qtisocketsink

Pad Name	Capabilities	
SINK template: 'sink' ▪ *Availability:* Always ▪ *Direction:* sink	ANY	–

Element configuration

Table 6-28 Element properties for qtisocketsink

Property	Description
name	The name of the object. ▪ flags: readable, writable ▪ String. Default: "fdsocketsink0"
parent	The parent of the object. ▪ flags: readable, writable ▪ Object of type "GstObject"
sync	Sync on the clock.

Table 6-28 Element properties for qtisocketsink (cont.)

Property	Description
	■ flags: readable, writable ■ Boolean. Default: true
max-lateness	Maximum number of nanoseconds that a buffer can be late before it's dropped (-1 unlimited). ■ flags: readable, writable ■ Integer64. Range: -1 to 9223372036854775807, default: -1
qos	Generate QoS events upstream. ■ flags: readable, writable ■ Boolean. Default: false
async	Go asynchronously to PAUSED. ■ flags: readable, writable ■ Boolean. Default: true
ts-offset	Timestamp offset in nanoseconds. ■ flags: readable, writable ■ Integer64. Range: -9223372036854775808 to 9223372036854775807, default: 0
enable-last-sample	Enable the last-sample property. ■ flags: readable, writable ■ Boolean. Default: true
last-sample	The last sample that was received in the sink. ■ flags: readable ■ Boxed pointer of type "GstSample"
blocksize	Size in bytes to pull per buffer (0 = default). ■ flags: readable, writable ■ Unsigned Integer. Range: 0 - 4294967295 Default: 4096
render-delay	More render delay of the sink in nanoseconds. ■ flags: readable, writable ■ Unsigned Integer64. Range: 0 to 18446744073709551615, default: 0
throttle-time	The time to keep between rendered buffers (0 = disabled). ■ flags: readable, writable ■ Unsigned Integer64. Range: 0 to 18446744073709551615, default: 0
max-bitrate	The maximum bits per second to render (0 = disabled). ■ flags: readable, writable ■ Unsigned Integer64. Range: 0 to 18446744073709551615, default: 0
socket	Location of the Unix domain socket. ■ flags: readable, writable, changeable only in NULL or READY state ■ String. Default: null

6.2.1.7 qtisocketsrc

The qtisocketsrc plugin uses UNIX sockets to receive FD backed GstBuffers from other processes that have qtisocketsink as the exit point.

The qtisocketsrc plugin requires a socket file (which should have the extension of .sock) to be passed as the socket property so it can receive the file descriptor.

After connecting to a socket, the plugin polls or waits to receive a GstBuffer that's used to create a block to contain the frame data received.

Customizations can be done on buffer such as:

- Block size: The read buffer size in bytes.
- num-buffers: The number of buffers that are sent to the output before sending the EOF signal.

Inheritance chain

GObject → GstObject → GstElement → GstBaseSrc → GstPushSrc → GstFdSocketSrc

The following tables provide information on pad templates and element properties of qtisocketsrc. For use cases, see Single camera stream with socket.

Pad configuration

Table 6-29 Pad templates for qtisocketsrc

Pad Name	Capabilities	
SRC template: 'src' ▪ *Availability:* Always ▪ *Direction:* source	ANY	–

Element configuration

Table 6-30 Element properties of qtisocketsrc

Property	Description
name	The name of the object. ▪ flags: readable, writable ▪ String. Default: "fdsocketsrc0"
parent	The parent of the object. ▪ flags: readable, writable ▪ Object of type "GstObject"
blocksize	Size in bytes to read per buffer (-1 = default). ▪ flags: readable, writable ▪ Unsigned Integer. Range: 0 to 4294967295, default: 4096
num-buffers	Number of buffers to output before sending EOS (-1 = unlimited). ▪ flags: readable, writable ▪ Integer. Range: -1 to 2147483647, default: -1

Table 6-30 Element properties of qtisocketsrc (cont.)

Property	Description
typefind	Run type-find before negotiating (deprecated, non-functional). ▪ flags: readable, writable, deprecated ▪ Boolean. Default: false
do-timestamp	Apply current stream time to buffers. ▪ flags: readable, writable ▪ Boolean. Default: false
socket	Location of the UNIX domain socket. ▪ flags: readable, writable, changeable only in NULL or READY state ▪ String. Default: null
timeout	Socket post timeout. ▪ flags: readable, writable, changeable only in NULL or READY state ▪ Unsigned Integer64. Range: 0 to18446744073709551615, default: 0

6.2.1.8 v4l2h264dec

The v4l2h264dec plugin uses the V4L2 APIs to decode the video stream.

- The plugin uses the H264 video decoder to provide hardware accelerated H.264 (MPEG-4 Part 10) decoding on supported platforms.
- The plugin is based on the GstVideoDecoder base class.

Inheritance chain

GObject → GstObject → GstElement → GstVideoDecoder → GstV4l2VideoDec → v4l2h264dec

The following tables provide information on pad templates and element properties of v4l2h264dec. For use cases, see Video playback use cases.

Pad configuration

Table 6-31 Pad templates for v4l2h264dec

Pad Name	Capabilities		
SINK template: 'sink' ▪ *Availability:* Always ▪ *Direction:* sink	video/x-h264	alignment:	au
		stream-format:	byte-stream
		level:	{ (string)1, (string)1b, (string)1.1, (string)1.2, (string)1.3, (string)2, (string)2.1, (string)2.2, (string)3, (string)3.1, (string)3.2, (string)4, (string)4.1, (string)4.2, (string)5, (string)5.1 }
		profile:	{ (string)baseline, (string)constrained-baseline, (string)main, (string)high, (string)constrained-high }
SRC template: 'src' ▪ *Availability:* Always ▪ *Direction:* source	video/x-raw	format:	{ (string)NV12, (string)NV21 }
		width:	[1, 32768]

Table 6-31 Pad templates for v4l2h264dec (cont.)

Pad Name	Capabilities		
		height:	[1, 32768]
		framerate:	[0/1, 2147483647/1]

Element configuration

NOTE For the current release, in the read-write properties, only **capture-io-mode** and **output-io-mode** are supported, and they're supported only with **(5): dmabuf-import - GST_V4L2_IO_DMABUF_IMPORT** property.

Table 6-32 Element properties of v4l2h264dec

Property	Description
automatic-request-sync-point-flags:	Flags to use when automatically requesting sync points. ■ flags: readable, writable ■ Flags "GstVideoDecoderRequestSyncPointFlags" Default: 0x00000003, "corrupt-output +discard-input" ▫ (0x00000001): discard-input - GST_VIDEO_DECODER_REQUEST_SYNC_POINT_DISCARD_INPUT ▫ (0x00000002): corrupt-output - GST_VIDEO_DECODER_REQUEST_SYNC_POINT_CORRUPT_OUTPUT
automatic-request-sync-points	Automatically request sync points when required. ■ flags: readable, writable ■ Boolean. Default: false
capture-io-mode	Capture I/O mode (matches src pad). ■ flags: readable, writable ■ Enum "GstV4l2IOMode" Default: 0, "auto" ▫ (0): auto - GST_V4L2_IO_AUTO ▫ (1): rw - GST_V4L2_IO_RW ▫ (2): mmap - GST_V4L2_IO_MMAP ▫ (3): userptr - GST_V4L2_IO_USERPTR ▫ (4): dmabuf - GST_V4L2_IO_DMABUF ▫ (5): dmabuf-import - GST_V4L2_IO_DMABUF_IMPORT
device	Device location. ■ flags: readable ■ String. Default: "/dev/video0"
device-fd	File descriptor of the device. ■ flags: readable ■ Integer. Range: -1 to 2147483647, default: -1
device-name	Name of the device. ■ flags: readable ■ String. Default: null

Table 6-32 Element properties of v4l2h264dec (cont.)

Property	Description
discard-corrupted-frames	Discard frames marked as corrupted instead of processing them as the output. ▪ flags: readable, writable ▪ Boolean. Default: false
extra-controls	Extra v4l2 controls (CIDs) for the device. ▪ flags: readable, writable ▪ Boxed pointer of type "GstStructure"
max-errors	Maximum consecutive decoder errors before returning a flow error. ▪ flags: readable, writable ▪ Integer. Range: -1 to 2147483647, default: 10
min-force-key-unit-interval	Minimum interval between force-key-unit requests in nanoseconds. ▪ flags: readable, writable ▪ Unsigned Integer64. Range: 0 - 18446744073709551615 Default: 0
name	The name of the object. ▪ flags: readable, writable, 0x2000 ▪ String. Default: "v4l2h264dec0"
output-io-mode	Output side I/O mode (matches sink pad). ▪ flags: readable, writable ▪ Enum "GstV4l2IOMode" Default: 0, "auto" ▫ (0): auto - GST_V4L2_IO_AUTO ▫ (1): rw - GST_V4L2_IO_RW ▫ (2): mmap - GST_V4L2_IO_MMAP ▫ (3): userptr - GST_V4L2_IO_USERPTR ▫ (4): dmabuf - GST_V4L2_IO_DMABUF ▫ (5): dmabuf-import - GST_V4L2_IO_DMABUF_IMPORT
parent	The parent of the object. ▪ flags: readable, writable, 0x2000 ▪ Object of type "GstObject"
qos	Handle quality-of-service events from downstream. ▪ flags: readable, writable ▪ Boolean. Default: true

6.2.1.9 v4l2h265dec

The v4l2h265dec plugin uses the V4L2 APIs to decode the video stream.

- The plugin uses the H.265 video decoder to provide hardware accelerated H.265 (MPEG-H Part 2) decoding on supported platforms.
- The plugin is based on the GstVideoDecoder base class.

Inheritance chain

GObject → GstObject → GstElement → GstVideoDecoder → GstV4l2VideoDec → v4l2h265dec

The following tables provide information on pad templates and element properties of v4l2h265dec. For use cases, see Video playback use cases.

Pad configuration

Table 6-33 Pad templates for v4l2h265dec

Pad Name	Capabilities		
SINK template: 'sink' ▪ *Availability:* Always ▪ *Direction:* sink	video/x-h265	alignment:	au
		stream-format:	byte-stream
SRC template: 'src' ▪ *Availability:* Always ▪ *Direction:* source	video/x-raw	format:	{ (string)NV12, (string)NV21 }
		width:	[1, 32768]
		height:	[1, 32768]
		framerate:	[0/1, 2147483647/1]

Element configuration

NOTE For the current release, in the read-write properties, only **capture-io-mode** and **output-io-mode** are supported, and they're supported only with **(5): dmabuf-import - GST_V4L2_IO_DMABUF_IMPORT** property.

Table 6-34 Element properties of v4l2h265dec

Property	Description
automatic-request-sync-point-flags:	Flags to use when automatically requesting sync points. ▪ flags: readable, writable ▪ Flags "GstVideoDecoderRequestSyncPointFlags" Default: 0x00000003, "corrupt-output +discard-input" ▫ (0x00000001): discard-input - GST_VIDEO_DECODER_REQUEST_SYNC_POINT_DISCARD_INPUT ▫ (0x00000002): corrupt-output - GST_VIDEO_DECODER_REQUEST_SYNC_POINT_CORRUPT_OUTPUT
automatic-request-sync-points	Automatically request sync points when required. ▪ flags: readable, writable ▪ Boolean. Default: false
capture-io-mode	Capture I/O mode (matches src pad). ▪ flags: readable, writable ▪ Enum "GstV4l2IOMode" Default: 0, "auto" ▫ (0): auto - GST_V4L2_IO_AUTO ▫ (1): rw - GST_V4L2_IO_RW ▫ (2): mmap - GST_V4L2_IO_MMAP ▫ (3): userptr - GST_V4L2_IO_USERPTR ▫ (4): dmabuf - GST_V4L2_IO_DMABUF ▫ (5): dmabuf-import - GST_V4L2_IO_DMABUF_IMPORT

Table 6-34 Element properties of v4l2h265dec (cont.)

Property	Description
device	Device location. ■ flags: readable ■ String. Default: "/dev/video0"
device-fd	File descriptor of the device. ■ flags: readable ■ Integer. Range: -1 to 2147483647, default: -1
device-name	Name of the device. ■ flags: readable ■ String. Default: null
discard-corrupted-frames	Discard frames marked as corrupted instead of processed them as the output. ■ flags: readable, writable ■ Boolean. Default: false
extra-controls	Extra v4l2 controls (CIDs) for the device. ■ flags: readable, writable ■ Boxed pointer of type "GstStructure"
max-errors	Maximum consecutive decoder errors before returning a flow error. ■ flags: readable, writable ■ Integer. Range: -1 to 2147483647, default: 10
min-force-key-unit-interval	Minimum interval between force-key-unit requests in nanoseconds. ■ flags: readable, writable ■ Unsigned Integer64. Range: 0 to 18446744073709551615, default: 0
name	The name of the object. ■ flags: readable, writable, 0x2000 ■ String. Default: "v4l2h265dec0"
output-io-mode	Output side I/O mode (matches sink pad). ■ flags: readable, writable ■ Enum "GstV4l2IOMode" Default: 0, "auto" □ (0): auto - GST_V4L2_IO_AUTO □ (1): rw - GST_V4L2_IO_RW □ (2): mmap - GST_V4L2_IO_MMAP □ (3): userptr - GST_V4L2_IO_USERPTR □ (4): dmabuf - GST_V4L2_IO_DMABUF □ (5): dmabuf-import - GST_V4L2_IO_DMABUF_IMPORT
parent	The parent of the object. ■ flags: readable, writable, 0x2000 ■ Object of type "GstObject"
qos	Handle Quality-of-Service events from downstream. ■ flags: readable, writable ■ Boolean. Default: true

6.2.1.10 v4l2h264enc

The v4l2h264enc plugin uses the V4L2 APIs to encode the video stream.

- The plugin uses the H.264 video encoder to provide hardware accelerated H.264 (MPEG-4 Part 10) encoding on supported platforms.
- The plugin is based on the GstVideoEncoder base class.

Inheritance chain

GObject → GstObject → GstElement → GstVideoEncoder → GstV4l2VideoEnc → v4l2h264enc

The following tables provide information on pad templates and element properties of v4l2h264enc. For use cases, see Video encode and decode and Slice-based encoding.

Pad configuration

Table 6-35 Pad templates for v4l2h264enc

Pad Name	Capabilities		
SINK template: 'sink' ▪ *Availability:* Always ▪ *Direction:* sink	video/x-raw	format:	{ (string)NV12, (string)NV21 }
		width:	[1, 32768]
		height:	[1, 32768]
		framerate:	[0/1, 2147483647/1]
SRC template: 'src' ▪ *Availability:* Always ▪ *Direction:* source	video/x-h264	alignment:	au
		stream-format:	byte-stream
		level:	{ (string)1, (string)1b, (string)1.1, (string)1.2, (string)1.3, (string)2, (string)2.1, (string)2.2, (string)3, (string)3.1, (string)3.2, (string)4, (string)4.1, (string)4.2, (string)5, (string)5.1 }
		profile:	{ (string)baseline, (string)constrained-baseline, (string)main, (string)high, (string)constrained-high }

Element configuration

Table 6-36 Element properties of v4l2h264enc

Property	Description
capture-io-mode	Capture I/O mode (matches src pad). ▪ flags: readable, writable ▪ Enum "GstV4l2IOMode"; default: 0, "auto" ▫ (0): auto - GST_V4L2_IO_AUTO ▫ (1): rw - GST_V4L2_IO_RW ▫ (2): mmap - GST_V4L2_IO_MMAP ▫ (3): userptr - GST_V4L2_IO_USERPTR

Table 6-36 Element properties of v4l2h264enc (cont.)

Property	Description
	▫ (4): dmabuf - GST_V4L2_IO_DMABUF ▫ (5): dmabuf-import - GST_V4L2_IO_DMABUF_IMPORT
device	Device location. ▪ flags: readable ▪ String. Default: "/dev/video1"
device-fd	File descriptor of the device. ▪ flags: readable ▪ Integer. Range: -1 to 2147483647, default: -1
device-name	Name of the device. ▪ flags: readable ▪ String. Default: null
extra-controls	Extra v4l2 control IDs (CIDs) for the device. ▪ flags: readable, writable ▪ Boxed pointer of type "GstStructure"
min-force-key-unit-interval	Minimum interval between force-key unit requests in nanoseconds. ▪ flags: readable, writable ▪ Unsigned Integer 64 ▪ Range: 0 - 18446744073709551615; default: 0
name	The name of the object. ▪ flags: readable, writable, 0x2000 ▪ String. Default: "v4l2h264enc0"
output-io-mode	Output side I/O mode (matches sink pad). ▪ flags: readable, writable ▪ Enum "GstV4l2IOMode" Default: 0, "auto" ▫ (0): auto - GST_V4L2_IO_AUTO ▫ (1): rw - GST_V4L2_IO_RW ▫ (2): mmap - GST_V4L2_IO_MMAP ▫ (3): userptr - GST_V4L2_IO_USERPTR ▫ (4): dmabuf - GST_V4L2_IO_DMABUF ▫ (5): dmabuf-import - GST_V4L2_IO_DMABUF_IMPORT
parent	The parent of the object. ▪ flags: readable, writable, 0x2000 ▪ Object of type "GstObject"
qos	Handle Quality-of-Service events from downstream. ▪ flags: readable, writable ▪ Boolean. Default: true

Table 6-37 v4l2ctl extra-controls–user controls

Control name	Control type	Parameters
horizontal_flip	Boolean	▪ value=0; this is the default value ▪ flags=execute-on-write
vertical_flip	Boolean	▪ value=0; this is the default value ▪ flags= execute-on-write
rotate	Integer	▪ min=0 ▪ max=270 ▪ step=90 ▪ value=0; this is the default value ▪ flags=execute-on-write and modify-layout

Table 6-38 v4l2ctl extra-controls–codec controls

Control name	Control type	Parameters
video_b_frames	Integer	▪ min=0 ▪ max=7 ▪ step=1 ▫ value=0; this is the default value ▫ flags=update and execute-on-write
video_gop_size	Integer	▪ min=0 ▪ max=2147483647 ▪ step=1 ▪ value=59; this is the default value ▪ flags=execute-on-write
video_bitrate_mode	Menu	▪ min=0 ▪ max=3 ▪ value=0: Variable Bitrate ▫ 0: Variable Bitrate; this is the default value ▫ 1: Constant Bitrate ▫ 3: Maximum Bitrate ▪ flags=update, execute-on-write
video_bitrate	Integer	▪ min=1 ▪ max=100000000 ▪ step=1 ▪ value=20000000 ▪ flags=execute-on-write ▪
frame_level_rate_control_enable	Integer	▪ value=1; this is the default value ▪ flags=execute-on-write
maximum_bytes_in_a_slice	Integer	▪ min=512 ▪ max=12500000 ▪ step=1

Table 6-38 v4l2ctl extra-controls–codec controls (cont.)

Control name	Control type	Parameters
		▪ value=512; this is the default value ▪ flags=execute-on-write
number_of_mbs_in_a_slice	Integer	▪ min=1 ▪ max=36864 ▪ step=1 ▪ value=1; this is the default value ▪ flags=execute-on-write
slice_partitioning_method	Menu	▪ min=0 ▪ max=2 ▪ value=0:Single ▫ 0: Single; this is the default value ▫ 1: Max macro blocks ▫ 2: Max Bytes ▪ flags=execute-on-write
ltr_count	Integer	▪ min=0 ▪ max=2 ▪ step=1 ▪ value=0; this is the default value ▪ flags=execute-on-write
frame_ltr_index	Integer	▪ min=-1 ▪ max=1 ▪ step=1 ▪ value=-1; this is the default value ▪ flags=execute-on-write
use_ltr_frames	Bitmask	▪ max=0x00000003 ▪ default=0x00000000 ▪ value=0 ▪ flags=execute-on-write
intra_refresh_period	Integer	▪ min=0 ▪ max=2147483647 ▪ step=1 ▪ value=0; this is the default value ▪ flags=execute-on-write
intra_refresh_period_type	Menu	▪ min=0 ▪ max=1 ▪ value=0 ▫ 0: Random; this is the default value ▫ 1: Cyclic ▪ flags=execute-on-write

Table 6-39 v4l2ctl extra-controls–H.264 codec controls

Control name	Control type	Parameters
h264_i_frame_qp_value	Integer	■ min=1 ■ max=51 ■ step=1 ■ value=20; this is the default value ■ flags=execute-on-write
h264_p_frame_qp_value	Integer	■ min=1 ■ max=51 ■ step=1 ■ value=20; this is the default value ■ flags=execute-on-write
h264_b_frame_qp_value	Integer	■ min=1 ■ max=51 ■ step=1 ■ value=20; this is the default value ■ flags=execute-on-write
h264_minimum_qp_value	Integer	■ min=1 ■ max=51 ■ step=1 ■ value=1; this is the default value ■ flags=execute-on-write
h264_maximum_qp_value	Integer	■ min=1 ■ max=51 ■ step=1 ■ value=51; this is the default value ■ flags=execute-on-write
h264_entropy_mode	Menu	■ min=0 ■ max=1 ■ value=1 (CABAC) □ 0: CAVLC □ 1: CABAC; this is the default value ■ flags=execute-on-write
h264_level	Menu	■ min=0 ■ max=15 ■ value=15 (5.1) □ 0: 1 □ 1: 1b □ 2: 1.1 □ 3: 1.2 □ 4: 1.3 □ 5: 2 □ 6: 2.1

Table 6-39 v4l2ctl extra-controls–H.264 codec controls (cont.)

Control name	Control type	Parameters
		▫ 7: 2.2 ▫ 8: 3 ▫ 9: 3.1 ▫ 10: 3.2 ▫ 11: 4 ▫ 12: 4.1 ▫ 13: 4.2 ▫ 14: 5 ▫ 15: 5.1; this is the default value ■ flags=execute-on-write
h264_profile	Menu	■ min=0 ■ max=17 ■ value=4 (High) ▫ 0: Baseline ▫ 1: Constrained Baseline ▫ 2: Main ▫ 4: High; this is the default value ▫ 17: Constrained High ■ flags=execute-on-write
enable_h264_hierarchical_coding	Integer	■ min=0 ■ max=1 ■ step=1 ■ value=0; this is the default value ■ flags=execute-on-write
h264_hierarchical_coding_type	Menu	■ min=0 ■ max=1 ■ value =1 ▫ 0: Hier Coding B ▫ 1: Hier Coding P; this is the default value ■ flags=execute-on-write
h264_number_of_hc_layers	Integer	■ min=0 ■ max=5 ■ step=1 ■ value=0; this is the default value ■ flags=execute-on-write
h264_i_frame_minimum_qp_value	Integer	■ min=1 ■ max=51 ■ step=1 ■ value=1; this is the default value ■ flags=execute-on-write
h264_i_frame_maximum_qp_value	Integer	■ min=1 ■ max=51

Table 6-39 v4l2ctl extra-controls–H.264 codec controls (cont.)

Control name	Control type	Parameters
		▪ step=1 ▪ value=51; this is the default value ▪ flags=execute-on-write
h264_p_frame_minimum_qp_value	Integer	▪ min=1 ▪ max=51 ▪ step=1 ▪ value=1; this is the default value ▪ flags=execute-on-write
h264_p_frame_maximum_qp_value	Integer	▪ min=1 ▪ max=51 ▪ step=1 ▪ value=51; this is the default value ▪ flags=execute-on-write
h264_b_frame_minimum_qp_value	Integer	▪ min=1 ▪ max=51 ▪ step=1 ▪ value=1; this is the default value ▪ flags=execute-on-write
h264_b_frame_maximum_qp_value	Integer	▪ min=1 ▪ max=51 ▪ step=1 ▪ value=51; this is the default value ▪ flags=execute-on-write
h264_hierarchical_lay_0_bitrate	Integer	▪ min=1 ▪ max=100000000 ▪ step=1 ▪ value=20000000; this is the default value ▪ flags=execute-on-write
h264_hierarchical_lay_1_bitrate	Integer	▪ min=1 ▪ max=100000000 ▪ step=1 ▪ value=20000000; this is the default value ▪ flags=execute-on-write
h264_hierarchical_lay_2_bitrate	Integer	▪ min=1 ▪ max=100000000 ▪ step=1 ▪ value=20000000; this is the default value ▪ flags=execute-on-write
h264_hierarchical_lay_3_bitrate	Integer	▪ min=1 ▪ max=100000000 ▪ step=1

Table 6-39 v4l2ctl extra-controls–H.264 codec controls (cont.)

Control name	Control type	Parameters
		▪ value=20000000; this is the default value ▪ flags=execute-on-write
h264_hierarchical_lay_4_bitrate	Integer	▪ min=1 ▪ max=100000000 ▪ step=1 ▪ value=20000000; this is the default value ▪ flags=execute-on-write
h264_hierarchical_lay_5_bitrate	Integer	▪ min=1 ▪ max=100000000 ▪ step=1 ▪ value=20000000; this is the default value ▪ flags=execute-on-write

6.2.1.11 v4l2h265enc

The v4l2h265enc plugin uses V4L2 APIs for encoding of video stream.

- The plugin uses the H.265 video encoder to provide hardware accelerated H.265 (MPEG-H Part 2) encoding on supported platforms
- The plugin is based on GstVideoEncoder base class.

Inheritance chain

GObject → GstObject → GstElement → GstVideoEncoder → GstV4l2VideoEnc → v4l2h265enc

The following tables provide information on pad templates and element properties of v4l2h265enc. For use cases, see Video encode and decode and Slice-based encoding.

Pad configuration

Table 6-40 Pad templates for v4l2h265enc

Pad Name	Capabilities		
SINK template: 'sink' ▪ *Availability:* Always ▪ *Direction:* sink	video/x-raw	format:	{ (string)NV12, (string)NV21 }
		width:	[1, 32768]
		height:	[1, 32768]
		framerate:	[0/1, 2147483647/1]
SRC template: 'src' ▪ *Availability:* Always ▪ *Direction:* source	video/x-h265	alignment:	au
		stream-format:	byte-stream

Element configuration

Table 6-41 Element properties of v4l2h265enc

Property	Description
capture-io-mode	Capture I/O mode (matches src pad). ■ flags: readable, writable ■ Enum "GstV4l2IOMode" Default: 0, "auto" ▫ (0): auto - GST_V4L2_IO_AUTO ▫ (1): rw - GST_V4L2_IO_RW ▫ (2): mmap - GST_V4L2_IO_MMAP ▫ (3): userptr - GST_V4L2_IO_USERPTR ▫ (4): dmabuf - GST_V4L2_IO_DMABUF ▫ (5): dmabuf-import - GST_V4L2_IO_DMABUF_IMPORT
device	The device location. ■ flags: readable ■ String. Default: "/dev/video1"
device-fd	The file descriptor of the device. ■ flags: readable ■ Integer. Range: -1 to 2147483647, default: -1
device-name	The name of the device. ■ flags: readable ■ String. Default: null
extra-controls	The extra v4l2 controls (CIDs) for the device. ■ flags: readable, writable ■ Boxed pointer of type "GstStructure"
min-force-key-unit-interval	The minimum interval between force-key-unit requests in nanoseconds. ■ flags: readable, writable ■ Unsigned Integer64. Range: 0 - 18446744073709551615 Default: 0
name	The name of the object. ■ flags: readable, writable, 0x2000 ■ String. Default: "v4l2h265enc0"
output-io-mode	The output side I/O mode (matches sink pad). ■ flags: readable, writable ■ Enum "GstV4l2IOMode" Default: 0, "auto" ▫ (0): auto - GST_V4L2_IO_AUTO ▫ (1): rw - GST_V4L2_IO_RW ▫ (2): mmap - GST_V4L2_IO_MMAP ▫ (3): userptr - GST_V4L2_IO_USERPTR ▫ (4): dmabuf - GST_V4L2_IO_DMABUF ▫ (5): dmabuf-import - GST_V4L2_IO_DMABUF_IMPORT

Table 6-41 Element properties of v4l2h265enc (cont.)

Property	Description
parent	The parent of the object. ▪ flags: readable, writable, 0x2000 ▪ Object of type "GstObject"
qos	Handle quality-of-service events from downstream. ▪ flags: readable, writable ▪ Boolean. Default: true

The table lists the codec controls for H.265. For the general v4l2ctl user and codec controls, see Table 6-37 and Table 6-38.

Table 6-42 v4l2ctl–H.265 codec controls

Control name	Control type	Parameters
hevc_minimum_qp_value	Integer	▪ min=1 ▪ max=51 ▪ step=1 ▪ value=1; this is the default value ▪ flags=execute-on-write
hevc_maximum_qp_value	Integer	▪ min=1 ▪ max=51 ▪ step=1 ▪ value=51; this is the default value ▪ flags=execute-on-write
hevc_i_frame_qp_value	Integer	▪ min=1 ▪ max=51 ▪ step=1 ▪ value=20; this is the default value ▪ flags=execute-on-write
hevc_p_frame_qp_value	Integer	▪ min=1 ▪ max=51 ▪ step=1 ▪ value=20; this is the default value ▪ flags=execute-on-write
hevc_b_frame_qp_value	Integer	▪ min=1 ▪ max=51 ▪ step=1 ▪ value=20; this is the default value ▪ flags=execute-on-write
hevc_hierarchical_coding_type	Menu	▪ min=0 ▪ max=1

Table 6-42 v4l2ctl–H.265 codec controls (cont.)

Control name	Control type	Parameters
		■ value=1 (P) ▫ 0: B ▫ 1: P; this is the default value ■ flags=execute-on-write
hevc_hierarchical_coding_layer	Integer	■ min=0 ■ max=5 ■ step=1 ■ value=0; this is the default value ■ flags=execute-on-write
hevc_profile	Menu	■ min=0 ■ max=2 ■ default=0 ■ value=0 (Main) ▫ 0: Main; this is the default value ▫ 1: Main Still Picture ▫ 2: Main 10 ■ flags=execute-on-write
hevc_level	Menu	■ min=0 ■ max=7 ■ value=7 (5) ▫ 0: 1 ▫ 1: 2 ▫ 2: 2.1 ▫ 3: 3 ▫ 4: 3.1 ▫ 5: 4 ▫ 6: 4.1 ▫ 7: 5; this is the default value ■ flags=execute-on-write
hevc_hierarchical_lay_0_bitrate	Integer	■ min=1 ■ max=100000000 ■ step=1 ■ value=20000000; this is the default value ■ flags=execute-on-write
hevc_hierarchical_lay_1_bitrate	Integer	■ min=1 ■ max=100000000 ■ step=1 ■ default=20000000; this is the default value ■ flags=execute-on-write

Table 6-42 v4l2ctl–H.265 codec controls (cont.)

Control name	Control type	Parameters
hevc_hierarchical_lay_2_bitrate	Integer	▪ min=1 ▪ max=100000000 ▪ step=1 ▪ value=20000000; this is the default value ▪ flags=execute-on-write
hevc_hierarchical_lay_3_bitrate	Integer	▪ min=1 ▪ max=100000000 ▪ step=1 ▪ value=20000000; this is the default value ▪ flags=execute-on-write
hevc_hierarchical_lay_4_bitrate	Integer	▪ min=1 ▪ max=100000000 ▪ step=1 ▪ value=20000000; this is the default value ▪ flags=execute-on-write
hevc_hierarchical_lay_5_bitrate	Integer	▪ min=1 ▪ max=100000000 ▪ step=1 ▪ value=20000000; this is the default value ▪ flags=execute-on-write
hevc_i_frame_minimum_qp_value	Integer	▪ min=1 ▪ max=51 ▪ step=1 ▪ value=1; this is the default value ▪ flags=execute-on-write
hevc_i_frame_maximum_qp_value	Integer	▪ min=1 ▪ max=51 ▪ step=1 ▪ value=51; this is the default value ▪ flags=execute-on-write
hevc_p_frame_minimum_qp_value	Integer	▪ min=1 ▪ max=51 ▪ step=1 ▪ value=1; this is the default value ▪ flags=execute-on-write
hevc_p_frame_maximum_qp_value	Integer	▪ min=1 ▪ max=51 ▪ step=1

Table 6-42 v4l2ctl–H.265 codec controls (cont.)

Control name	Control type	Parameters
		▪ value=51; this is the default value ▪ flags=execute-on-write
hevc_b_frame_minimum_qp_value	Integer	▪ min=1 ▪ max=51 ▪ step=1 ▪ value=1; this is the default value ▪ flags=execute-on-write
hevc_b_frame_maximum_qp_value	Integer	▪ min=1 ▪ max=51 ▪ step=1 ▪ value=51; this is the default value ▪ flags=execute-on-write

6.2.1.12 v4l2vp9dec

The v4l2vp9dec plugin uses the V4L2 APIs to decode the video stream.

This plugin uses the VP9 video decoder to provide hardware-accelerated VP9 decoding on platforms that support it. It's based on the GstVideoDecoder base class.

Inheritance chain

GObject → GstObject → GstElement → GstVideoDecoder → GstV4l2VideoDec → v4l2vp9dec

The following tables provide information on pad templates and element properties of v4l2vp9dec. For use cases, see Video playback use cases.

Pad configuration

Table 6-43 Pad templates for v4l2vp9dec

Pad Name	Capabilities		
SINK template: 'sink' ▪ *Availability:* Always ▪ *Direction:* sink	video/x-vp9		
SRC template: 'src' ▪ *Availability:* Always ▪ *Direction:* source	video/x-raw	format:	{ (string)NV12, (string)NV21 }
		width:	[1, 32768]
		height:	[1, 32768]
		framerate:	[0/1, 2147483647/1]

Element configuration

NOTE For the current release, in the read/write properties, only **capture-io-mode** and **output-io-mode** are supported with **(5): dmabuf-import - GST_V4L2_IO_DMABUF_IMPORT** property.

Table 6-44 Element properties of v4l2vp9dec

Property	Description
automatic-request-sync-point-flags:	The flags to use when automatically requesting sync points. ■ flags: readable, writable ■ Flags "GstVideoDecoderRequestSyncPointFlags" Default: 0x00000003, "corrupt-output+discard-input" ▫ (0x00000001): discard-input - GST_VIDEO_DECODER_REQUEST_SYNC_POINT_DISCARD_INPUT ▫ (0x00000002): corrupt-output - GST_VIDEO_DECODER_REQUEST_SYNC_POINT_CORRUPT_OUTPUT
automatic-request-sync-points	Automatically request sync points when required. ■ flags: readable, writable ■ Boolean. Default: false
capture-io-mode	Capture I/O mode (matches src pad). ■ flags: readable, writable ■ Enum "GstV4l2IOMode" Default: 0, "auto" ▫ (0): auto - GST_V4L2_IO_AUTO ▫ (1): rw - GST_V4L2_IO_RW ▫ (2): mmap - GST_V4L2_IO_MMAP ▫ (3): userptr - GST_V4L2_IO_USERPTR ▫ (4): dmabuf - GST_V4L2_IO_DMABUF ▫ (5): dmabuf-import - GST_V4L2_IO_DMABUF_IMPORT
device	The device location. ■ flags: readable ■ String. Default: "/dev/video0"
device-fd	The file descriptor of the device. ■ flags: readable ■ Integer. Range: -1 to 2147483647, default: -1
device-name	The name of the device. ■ flags: readable ■ String. Default: null
discard-corrupted-frames	Discard frames marked as corrupted instead of outputting them. ■ flags: readable, writable ■ Boolean. Default: false
extra-controls	Extra v4l2 controls (CIDs) for the device. ■ flags: readable, writable ■ Boxed pointer of type "GstStructure"
max-errors	Maximum consecutive decoder errors before returning flow error. ■ flags: readable, writable ■ Integer. Range: -1 to 2147483647, default: 10
min-force-key-unit-interval	Minimum interval between force-key-unit requests in nanoseconds. ■ flags: readable, writable ■ Unsigned Integer64. Range: 0 to 18446744073709551615, default: 0

Table 6-44 Element properties of v4l2vp9dec (cont.)

Property	Description
name	The name of the object. ▪ flags: readable, writable, 0x2000 ▪ String. Default: "v4l2vp9dec0"
output-io-mode	Output side I/O mode (matches sink pad). ▪ flags: readable, writable ▪ Enum "GstV4l2IOMode" Default: 0, "auto" ▫ (0): auto - GST_V4L2_IO_AUTO ▫ (1): rw - GST_V4L2_IO_RW ▫ (2): mmap - GST_V4L2_IO_MMAP ▫ (3): userptr - GST_V4L2_IO_USERPTR ▫ (4): dmabuf - GST_V4L2_IO_DMABUF ▫ (5): dmabuf-import - GST_V4L2_IO_DMABUF_IMPORT
parent	The parent of the object. ▪ flags: readable, writable, 0x2000 ▪ Object of type "GstObject"
qos	Handle quality-of-service events from downstream. ▪ flags: readable, writable ▪ Boolean. Default: true

6.2.2 Configure audio plugins

The audio plugins allow you to capture and play the audio streams.

RELATED INFORMATION

"Configure display, camera, encode and decode plugins" on page 343
"Configure ML plugins" on page 397

6.2.2.1 pulsesrc

The pulsesrc plugin interacts with the underlying PulseAudio sound server to capture the pulse-coded modulation (PCM) samples.

The PulseAudio server interacts with the underlying hardware to provide the capture capability and is based on the GstAudioSrc base class.

Inheritance chain

GObject → GstObject → GstElement → GstBaseSrc → GstPushSrc → GstAudioBaseSrc → GstAudioSrc → GstPulseSrc

The following tables provide information on pad templates and element properties of pulsesrc. For use cases, see Audio (only) capture.

Pad configuration

Table 6-45 Pad templates for pulsesrc

Pad Name	Capabilities		
SRC template: 'src' ■ *Availability:* Always ■ *Direction:* src	audio/x-raw	format:	{ (string)S16LE, (string)S16BE, (string)F32LE, (string)F32BE, (string)S32LE, (string)S32BE, (string)S24LE, (string)S24BE, (string)S24_32LE, (string)S24_32BE, (string)U8 }
		layout:	interleaved
		rate:	[1, 22579200]
		channels:	[1, 32]
	audio/x-alaw	rate:	[1, 22579200]
		channels:	[1, 32]
	audio/x-mulaw	rate:	[1, 22579200]
		channels:	[1, 32]

Element configuration

Table 6-46 Element properties of pulsesrc

Property	Description
name	The name of the object. ■ flags: readable, writable ■ String. Default: "pulsesrc0"
parent	The parent of the object. ■ flags: readable, writable ■ Object of type "GstObject"
blocksize	Size in bytes to read per buffer (-1 = default). ■ flags: readable, writable ■ Unsigned Integer. Range: 0 - 4294967295 Default: 0
num-buffers	Number of buffers to output before sending EOS (-1 = unlimited). ■ flags: readable, writable ■ Integer. Range: -1 to 2147483647, default: -1
typefind	Run type find before negotiating (deprecated, non-functional). ■ flags: readable, writable, deprecated ■ Boolean. Default: false
do-timestamp	Apply current stream time to buffers. ■ flags: readable, writable ■ Boolean. Default: true
buffer-time	Size of audio buffer in microseconds. This size is the maximum amount of data that's buffered in the device and the maximum latency that the source reports. The pulsesrc plugin can ignore this value. ■ flags: readable, writable ■ Integer64. Range: 1 - 9223372036854775807 Default: 200000

Table 6-46 Element properties of pulsesrc (cont.)

Property	Description
latency-time	The minimum amount of data to read in each iteration is in microseconds. This is the minimum latency that the source reports. The plugin can ignore this value. ■ flags: readable, writable ■ Integer64. Range: 1 - 9223372036854775807 Default: 10000
actual-buffer-time	Actual configured size of audio buffer in microseconds. ■ flags: readable ■ Integer64. Range: -1 to 9223372036854775807, default: -1
actual-latency-time	Actual configured audio latency in microseconds. ■ flags: readable ■ Integer64. Range: -1 to 9223372036854775807, default: -1
provide-clock	Provide a clock to be used as the global pipeline clock. ■ flags: readable, writable ■ Boolean. Default: true
slave-method	Algorithm used to match the rate of the primary clock. ■ flags: readable, writable ■ Enum "GstAudioBaseSrcSlaveMethod" Default: 2, "skew" ▫ (0): resample - GST_AUDIO_BASE_SRC_SLAVE_RESAMPLE ▫ (1): re-timestamp - GST_AUDIO_BASE_SRC_SLAVE_RE_TIMESTAMP ▫ (2): skew - GST_AUDIO_BASE_SRC_SLAVE_SKEW ▫ (3): none - GST_AUDIO_BASE_SRC_SLAVE_NONE
server	The PulseAudio server for the connection. ■ flags: readable, writable ■ String. Default: null
device	The PulseAudio source device for connection. ■ flags: readable, writable ■ String. Default: null
current-device	The current PulseAudio source device. ■ flags: readable ■ String. Default: ""
device-name	Human-readable name of the sound device. ■ flags: readable ■ String. Default: null
volume	Linear volume of this stream, 1.0=100%. ■ flags: readable, writable ■ Double. Range: 0 - 10 Default: 1
mute	Mute state of this stream. ■ flags: readable, writable ■ Boolean. Default: false
client-name	The PulseAudio client_name_to_use.

Table 6-46 Element properties of pulsesrc (cont.)

Property	Description
	■ flags: readable, writable, changeable only in NULL or READY state ■ String. Default: "gst-inspect-1.0"
stream-properties	A list of PulseAudio stream properties. ■ flags: readable, writable ■ Boxed pointer of type "GstStructure"
source-output-index	The index of the PulseAudio source output corresponding to this record stream. ■ flags: readable ■ Unsigned Integer. Range: 0 - 4294967295 Default: 4294967295
stream-flags	Stream flags (not all default flags can be overridden). ■ flags: writable ■ Flags "GstPulseStreamFlags" Default: 0x0000000f, "Timing update requests are issued periodically and automatically. Don't force the time to increase monotonically and interpolate the latency for this stream. ▫ ((0x00000000): Flag to pass when no specific options are needed - GST_PULSE_STREAM_NOFLAGS ▫ (0x00000001): Create the corked stream- GST_PULSE_STREAM_START_CORKED ▫ (0x00000002): Interpolate the latency for this stream - GST_PULSE_STREAM_INTERPOLATE_TIMING ▫ (0x00000004): Don't force the time to increase monotonically - GST_PULSE_STREAM_NOT_MONOTONIC ▫ (0x00000008): Timing update requests are issued periodically and automatically - GST_PULSE_STREAM_AUTO_TIMING_UPDATE ▫ (0x00000010): Don't remap channels by their name, instead map them simply by their index - GST_PULSE_STREAM_NO_REMAP_CHANNELS ▫ (0x00000020): When remapping channels by name, don't upmix or downmix them to related channels - GST_PULSE_STREAM_NO_REMIX_CHANNELS ▫ (0x00000040): Use the sample format of the sink/device this stream is being connected to - GST_PULSE_STREAM_FIX_FORMAT ▫ (0x00000080): Use the sample rate of the sink - GST_PULSE_STREAM_FIX_RATE ▫ (0x00000100): Use the number of channels and the channel map of the sink - GST_PULSE_STREAM_FIX_CHANNELS ▫ (0x00000200): Don't allow moving of this stream to another sink/device - GST_PULSE_STREAM_DONT_MOVE ▫ (0x00000400): Allow dynamic changing of the sampling rate during playback - GST_PULSE_STREAM_VARIABLE_RATE ▫ (0x00000800): Find peaks instead of resampling - GST_PULSE_STREAM_PEAK_DETECT ▫ (0x00001000): Create in muted state - GST_PULSE_STREAM_START_MUTED ▫ (0x00002000): Try to adjust the latency of the sink/source based on the requested buffer metrics and adjust buffer metrics accordingly - GST_PULSE_STREAM_ADJUST_LATENCY ▫ (0x00004000): Enable compatibility mode for legacy clients that rely on a classic hardware device fragment-style playback model - GST_PULSE_STREAM_EARLY_REQUESTS

Table 6-46 Element properties of pulsesrc (cont.)

Property	Description
	▫ (0x00008000): If set, this stream isn't considered when it's checked whether the stream is connected. It's auto-suspended. GST_PULSE_STREAM_DONT_INHIBIT_AUTO_SUSPEND ▫ (0x00010000): Create in unmuted state - GST_PULSE_STREAM_START_UNMUTED ▫ (0x00020000): If the sink/source of this stream is connected, it's suspended during the creation of this stream, causing it to fail - GST_PULSE_STREAM_FAIL_ON_SUSPEND ▫ (0x00040000): If a volume is passed when this stream is created, creation of this stream causes it to fail - GST_PULSE_STREAM_RELATIVE_VOLUME ▫ ((0x00080000): Used to tag content that's rendered by pass-through sinks - GST_PULSE_STREAM_PASSTHROUGH write-only

6.2.2.2 pulsesink

The pulsesink plugin interacts with the underlying PulseAudio sound server to play the PCM samples.

The PulseAudio Server interacts with the underlying hardware to provide the playback capability. It's based on the GstAudioBaseSink base class.

Inheritance chain

GObject → GstObject → GstElement → GstBaseSink → GstAudioBaseSink → GstPulseSink

The following tables provide information on pad templates and element properties of pulsesink. For use cases, see Audio (only) playback.

Pad configuration

Table 6-47 Pad templates for pulsesink

Pad Name	Capabilities		
SINK template: 'sink' *Availability:* Always *Direction:* sink	audio/x-raw	format:	{ (string)S16LE, (string)S16BE, (string)F32LE, (string)F32BE, (string)S32LE, (string)S32BE, (string)S24LE, (string)S24BE, (string)S24_32LE, (string)S24_32BE, (string)U8 }
		layout:	interleaved
		rate:	[1, 22579200]
		channels:	[1, 32]
	audio/x-alaw	rate:	[1, 22579200]
		channels:	[1, 32]
	audio/x-mulaw	rate:	[1, 22579200]
		channels:	[1, 32]
	audio/x-ac3	framed:	true
	audio/x-eac3	framed:	true
	audio/x-dts	framed:	true

Table 6-47 Pad templates for pulsesink (cont.)

Pad Name	Capabilities		
		block-size:	{ (int)512, (int)1024, (int)2048 }
	audio/mpeg	mpegversion:	1
		mpegaudioversion:	[1, 3]
		parsed:	true
	audio/mpeg	mpegversion:	{ (int)2, (int)4 }
		framed:	true
		stream-format:	adts

Element configuration

Table 6-48 Elements properties of pulsesink

Property	Description
name	The name of the object. ▪ flags: readable, writable ▪ String. Default: "pulsesink0"
parent	The parent of the object. ▪ flags: readable, writable ▪ Object of type "GstObject"
sync	Sync on the clock. ▪ flags: readable, writable ▪ Boolean. Default: true
max-lateness	Maximum number of nanoseconds that a buffer can be late before it's dropped (-1 unlimited). ▪ flags: readable, writable ▪ Integer64. Range: -1to 9223372036854775807, default: -1
qos	Generate wuality-of-service events upstream. ▪ flags: readable, writable ▪ Boolean. Default: false
async	Go asynchronously to PAUSED. ▪ flags: readable, writable ▪ Boolean. Default: true
ts-offset	Timestamp offset in nanoseconds. ▪ flags: readable, writable ▪ Integer64. Range: -9223372036854775808 to 9223372036854775807, default: 0
enable-last-sample	Enable the last-sample property. ▪ flags: readable, writable ▪ Boolean. Default: false
last-sample	The last sample received in the sink. ▪ flags: readable ▪ Boxed pointer of type "GstSample"

Table 6-48 Elements properties of pulsesink (cont.)

Property	Description
blocksize	Size in bytes to pull per buffer (0 = default). ▪ flags: readable, writable ▪ Unsigned Integer. Range: 0 - 4294967295 Default: 4096
render-delay	Additional render delay of the sink in nanoseconds. ▪ flags: readable, writable ▪ Unsigned Integer64. Range: 0 - 18446744073709551615 Default: 0
throttle-time	The time to keep between rendered buffers (0 = disabled). ▪ flags: readable, writable ▪ Unsigned Integer64. Range: 0 - 18446744073709551615 Default: 0
max-bitrate	The maximum bits per second to render (0 = disabled). ▪ flags: readable, writable ▪ Unsigned Integer64. Range: 0 - 18446744073709551615 Default: 0
buffer-time	Size of audio buffer in microseconds, this is the minimum latency that the sink reports. ▪ flags: readable, writable ▪ Integer64. Range: 1 - 9223372036854775807 Default: 200000
latency-time	The minimum amount of data to write in each iteration in microseconds. ▪ flags: readable, writable ▪ Integer64. Range: 1 - 9223372036854775807 Default: 10000
provide-clock	Provide a clock to be used as the global pipeline clock. ▪ flags: readable, writable ▪ Boolean. Default: true
slave-method	Algorithm used to match the rate of the primary clock. ▪ flags: readable, writable Enum "GstAudioBaseSinkSlaveMethod" Default: 1, "skew" ▪ (0): resample - GST_AUDIO_BASE_SINK_SLAVE_RESAMPLE ▪ (1): skew - GST_AUDIO_BASE_SINK_SLAVE_SKEW ▪ (2): none - GST_AUDIO_BASE_SINK_SLAVE_NONE ▪ (3): custom - GST_AUDIO_BASE_SINK_SLAVE_CUSTOM
can-activate-pull	Allow pull-based scheduling. ▪ flags: readable, writable ▪ Boolean. Default: false
alignment-threshold	Timestamp alignment threshold in nanoseconds. ▪ flags: readable, writable ▪ Unsigned Integer64. Range: 1 - 18446744073709551614 Default: 40000000
drift-tolerance	Tolerance for clock drift in microseconds. ▪ flags: readable, writable ▪ Integer64. Range: 1 - 9223372036854775807 Default: 40000
discont-wait	Window of time in nanoseconds to wait before creating a discontinuity.

Table 6-48 Elements properties of pulsesink (cont.)

Property	Description
	■ flags: readable, writable ■ Unsigned Integer64. Range: 0 - 18446744073709551614 Default: 1000000000
server	The PulseAudio server for connection. ■ flags: readable, writable ■ String. Default: null
device	The PulseAudio sink device for connection. ■ flags: readable, writable ■ String. Default: null
current-device	The current PulseAudio sink device. ■ flags: readable ■ String. Default: ""
device-name	Human-readable name of the sound device. ■ flags: readable ■ String. Default: null
volume	Linear volume of this stream, 1.0=100%. ■ flags: readable, writable ■ Double. Range: 0 - 10 Default: 1
mute	Mute state of this stream. ■ flags: readable, writable ■ Boolean. Default: false
client-name	The PulseAudio client name. ■ flags: readable, writable, changeable only in NULL or READY state ■ String. Default: "gst-inspect-1.0"
stream-properties	The list of PulseAudio stream properties. ■ flags: readable, writable ■ Boxed pointer of type "GstStructure"
stream-flags	The stream flags to use (not all default flags can be overridden). ■ flags: writable ■ Flags "GstPulseStreamFlags" Default: 0x0000200b ■ Adjust the latency of the sink/source based on the requested buffer metrics and adjust buffer metrics accordingly. The timing update requests are issued periodically automatically. Interpolate the latency for this stream and create the corked stream. □ (0x00000000): Flag to pass when no specific options are needed - GST_PULSE_STREAM_NOFLAGS □ (0x00000001): Create the stream corked - GST_PULSE_STREAM_START_CORKED □ (0x00000002): Interpolate the latency for this stream - GST_PULSE_STREAM_INTERPOLATE_TIMING □ (0x00000004): Don't force the time to increase monotonically - GST_PULSE_STREAM_NOT_MONOTONIC □ (0x00000008): Timing update requests are issued periodically and automatically - GST_PULSE_STREAM_AUTO_TIMING_UPDATE □ (0x00000010): Don't remap channels by their name, instead map them by their index - GST_PULSE_STREAM_NO_REMAP_CHANNELS

Table 6-48 Elements properties of pulsesink (cont.)

Property	Description
	▫ (0x00000020): When remapping channels by name, don't upmix or downmix them to related channels - GST_PULSE_STREAM_NO_REMIX_CHANNELS ▫ (0x00000040): Use the sample format of the sink/device to which the stream being connected - GST_PULSE_STREAM_FIX_FORMAT ▫ (0x00000080): Use the sample rate of the sink - GST_PULSE_STREAM_FIX_RATE ▫ (0x00000100): Use the number of channels and the channel map of the sink - GST_PULSE_STREAM_FIX_CHANNELS ▫ (0x00000200): Don't allow moving of this stream to another sink/device - GST_PULSE_STREAM_DONT_MOVE ▫ (0x00000400): Allow dynamic changing of the sampling rate during playback - GST_PULSE_STREAM_VARIABLE_RATE ▫ (0x00000800): Find peaks instead of resampling - GST_PULSE_STREAM_PEAK_DETECT ▫ (0x00001000): Create in muted state - GST_PULSE_STREAM_START_MUTED ▫ (0x00002000): Adjust the latency of the sink/source based on the requested buffer metrics and adjust buffer metrics accordingly - GST_PULSE_STREAM_ADJUST_LATENCY ▫ (0x00004000): Enable the compatibility mode for legacy clients that rely on a classic hardware device fragment-style playback model - GST_PULSE_STREAM_EARLY_REQUESTS ▫ (0x00008000): If set, this stream isn't considered when the device this stream is connected to auto-suspends - GST_PULSE_STREAM_DONT_INHIBIT_AUTO_SUSPEND ▫ (0x00010000): Create in unmuted state - GST_PULSE_STREAM_START_UNMUTED ▫ (0x00020000): If the sink/source connected to this stream is suspended during the creation of the stream, it cause it to fail - GST_PULSE_STREAM_FAIL_ON_SUSPEND ▫ (0x00040000): If volume is passed when this stream is created, creation of this stream fails - GST_PULSE_STREAM_RELATIVE_VOLUME ▫ (0x00080000): Used to tag content that are rendered by passthrough sinks - GST_PULSE_STREAM_PASSTHROUGH Write only
prebuf	Prebuf to be set with sink-input. ■ flags: writable ■ Integer. Range: -1 to 2147483647, default: 0 write-only

6.2.3 Configure ML plugins

The ML plugins facilitate the preprocessing, inferencing, and postprocessing of the machine learning models.

RELATED INFORMATION

"Configure display, camera, encode and decode plugins" on page 343
"Configure audio plugins" on page 389

6.2.3.1 qtimlvconverter

The qtimlvconverter plugin transforms the incoming video buffers into neural-network tensors while performing necessary format conversion and resizing in the process. To achieve these operations, the plugin uses the GPU hardware and ION/DMA allocated buffers.

For floating point tensors, the mean and sigma properties are required to transform the 8-bit unsigned integer images from the video source using the formula:

(channel - mean) × sigma where channel is the value for Red, Green, Blue, or Alpha channels of the color converted input.

For example, the table lists mean and sigma properties that impact the channel transformation of an image:

Table 6-49 Channel transformation example

Properties	RGB image with channel transformation
R=255, G=127, B=0	R=(255 - 128.0) x 0.75, G=(127 - 156.0) x 0.34, B=(0 - 124.0) x 0.02
mean="<128.0, 156.0, 124.0, 58.0>"	
sigma="<0.75, 0.34, 0.02, 0.07>"	

The subpixel-layout property specifies the arrangement of the image pixels in the output tensor.

The plugin uses Qualcomm IB2C library for all conversion operations. This library is wrapped inside the custom GstGlesVideoConverter abstraction layer with APIs to create, configure, and process the incoming and outgoing buffers.

GstMLBufferPool, a custom buffer pool class, can do the following:

- Assign the output buffers.
- Assign ION buffers through IOCTL commands to the kernel.

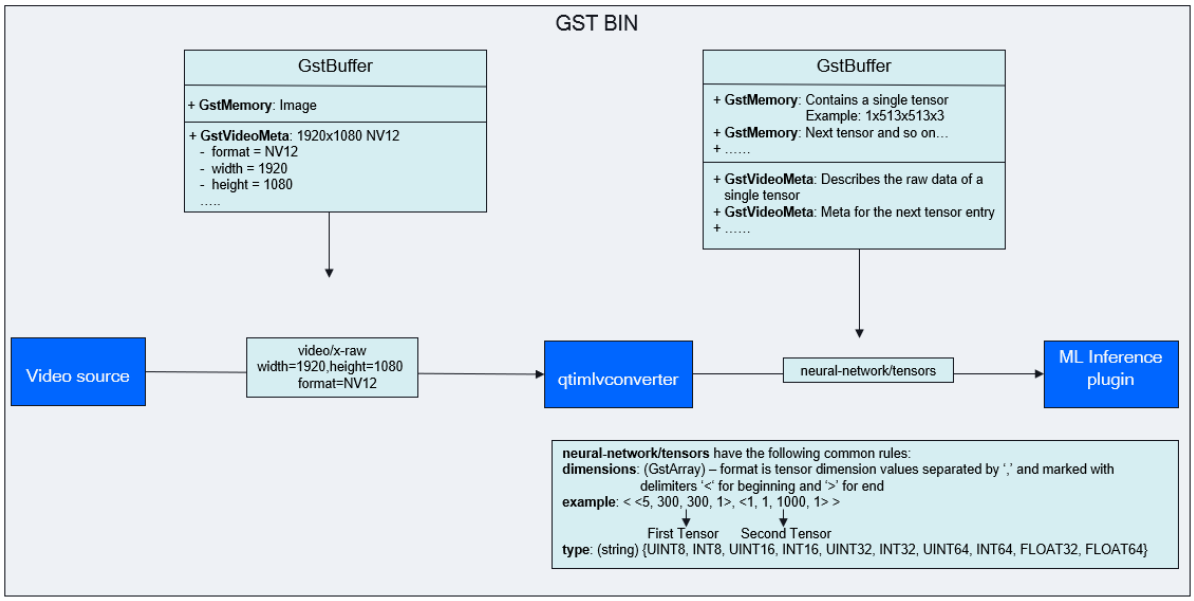


Figure 6-14 Gstbuffer workflow with qtimlvconverter

Inheritance chain

GObject → GstObject → GstElement → GstBaseTransform → GstMLVideoConverter

The following tables provide information on pad templates and element properties of qtimlvconverter. For use cases, see LiteRT use cases and Qualcomm Neural Processing SDK use cases.

Pad configuration

Table 6-50 Pad templates for qtimlvconverter

Pad Name	Capabilities		
SINK template: 'sink' *Availability:* Always *Direction:* sink	video/x-raw	format:	{ (string)RGBA, (string)BGRA, (string)ABGR, (string)ARGB, (string)RGBx, (string)BGRx, (string)xRGB, (string)xBGR, (string)BGR, (string)RGB, (string)GRAY8, (string)NV12, (string)NV21, (string)YUY2, (string)UYVY }
	video/x-raw(memory:GBM)	format:	{ (string)RGBA, (string)BGRA, (string)ABGR, (string)ARGB, (string)RGBx, (string)BGRx, (string)xRGB, (string)xBGR, (string)BGR, (string)RGB, (string)GRAY8, (string)NV12, (string)NV21, (string)YUY2, (string)UYVY }
SRC template: 'src' *Availability:* Always *Direction:* source	neural-network/tensors	type:	{ (string)UINT8, (string)INT32, (string)FLOAT16, (string)FLOAT32 }

Element configuration

Table 6-51 Element properties for qtimlvconverter

Property	Description
name	The name of the object. ▪ flags: readable, writable ▪ String. Default: "mlvideoconverter0"
parent	The parent of the object. ▪ flags: readable, writable ▪ Object of type "GstObject"
qos	Handle the quality-of-service events. ▪ flags: readable, writable ▪ Boolean. Default: false
subpixel-layout	Arrangement of the image pixels in the output tensor. ▪ flags: readable, writable ▪ Enum "GstMLVideoPixelLayout" Default: 0, "regular" ▫ (0): regular - Regular subpixel layout. For example, RGB, RGBA, or RGBx. ▫ (1): reverse - Reverse subpixel layout. For example, BGR, BGRA, or BGRx.
mean	Channels mean subtraction values for FLOAT tensors ('<R, G, B>', '<R, G, B, A>', '<G>'). ▪ flags: readable, writable ▪ GstValueArray of GValues of type "gdouble"

Table 6-51 Element properties for qtimlvconverter (cont.)

Property	Description
sigma	Channel divisor values for FLOAT tensors ('<R, G, B>', '<R, G, B, A>', '<G>'). ▪ flags: readable, writable ▪ GstValueArray of GValues of type "gdouble"
engine	Choose Qualcomm® Computer Vision SDK or GLES as the backend of qtimlvconverter. ▪ Value can be “fcv” or “gles” ▪ For Non-GPU device, the default value is fcv. For GPU device, the default value is gles **NOTE** The Qualcomm Computer Vision SDK (fcv) engine is currently not supported.

6.2.3.2 qtimlaconverter

The qtimlaconverter plugin processes the incoming audio waveform data into ML tensors. The ML models such as the audio classification model process these tensors for inferencing.

GstCaps determines the transformation on the plugin output. GstMLBufferPool, a custom buffer pool class, can do the following:

- Assign the output buffers.
- Assign the ION buffers through IOCTL commands to the kernel.

Inheritance chain

GObject → GstObject → GstElement → GstBaseTransform → GstMLAudioConverter

The following tables provide information on pad templates and element properties of qtimlsnpe. For use cases, Audio classification decode and display with LiteRT.

Pad configuration

Table 6-52 Pad templates for qtimlaconverter

Pad Name	Capabilities		
SINK template: 'sink' ▪ *Availability:* always ▪ *Direction:* sink	audio/x-raw	format	{ (string)F32LE, (string)S32LE, (string)U32LE, (string)S24LE, (string)U24LE, (string)S16LE, (string)U16LE, (string)S8, (string)U8 }
SRC template: 'src' ▪ *Availability:* Always ▪ *Direction:* source	neural-network/ tensors	format	{ (string)UINT8, (string)INT32, (string)FLOAT16, (string)FLOAT32 }

Element configuration

Table 6-53 Element properties of qtimlaconverter

Property	Description
name	The name of the object. ▪ flags: readable, writable ▪ String. Default: "mlaconverter0"
parent	The parent of the object. ▪ flags: readable, writable ▪ Object of type "GstObject"
qos	Handle the quality-of-service events. ▪ flags: readable, writable ▪ Boolean. Default: false
sample-rate	Sample-rate at which the audio converter receives the raw audio waveform samples. For example, the source sample rate when the audio source is an element pulsesrc.

6.2.3.3 qtibatch

The qtibatch plugin uses frame aggregation techniques to group several audio/video frames into one buffer for preprocessing. Batching can be done on a single stream (in which case the batch size is 1) or in many parallel streams.

- The loaded model in the ML inferencing plugin downstream determines the batch size.
- GstCaps shared the batch size with upstream.
- When creating the pipeline, the input sources must be lower or equal to the batch size. Inputs above the batch size are ignored.
- The plugin places the buffers from each source as separate GstMemory blocks along with their corresponding batch index.

  For example, source 0 at batch index 0 and source 1 at batch index 1.

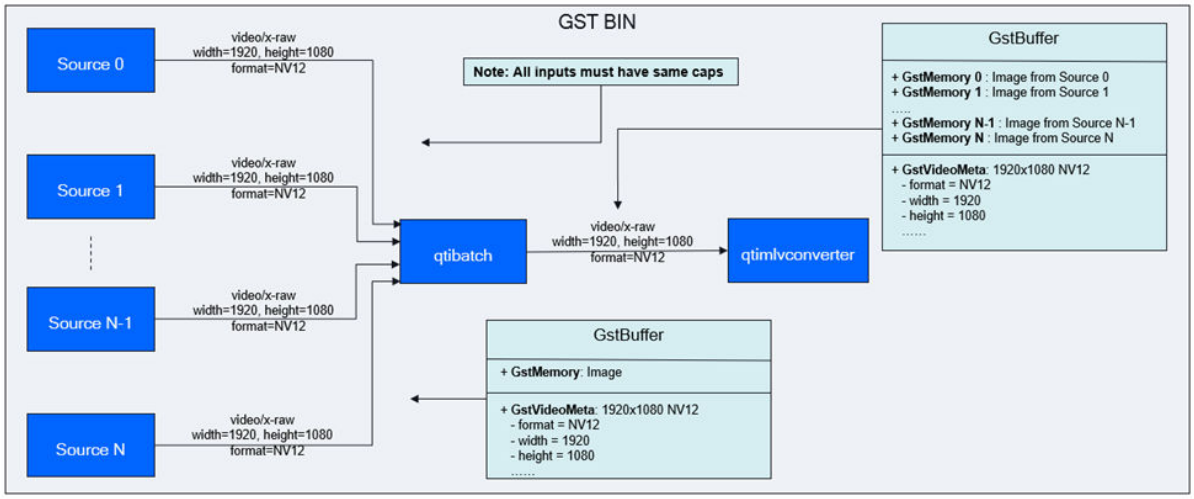


Figure 6-15 Gstbuffer workflow with qtibatch

Inheritance chain

GObject → GstObject → GstElement → GstBatch

Pad configuration

Table 6-54 Pad templates for qtibatch

Pad Name	Capabilities
SINK template: 'sink_%u' ▪ *Availability:* On request ▪ *Direction:* sink	video/x-raw (ANY)
	audio/x-raw (ANY)
SRC template: 'src' ▪ *Availability:* Always ▪ *Direction:* source	video/x-raw (ANY)
	audio/x-raw (ANY)

Element configuration

Table 6-55 Element properties for qtibatch

Property	Description
name	The name of the object. ▪ flags: readable, writable ▪ String. Default: "batch0"
parent	The parent of the object. ▪ flags: readable, writable ▪ Object of type "GstObject"

6.2.3.4 qtimlsnpe

The qtimlsnpe plugin shows the Qualcomm® Neural Processing SDK capabilities (load and run the models).

It accepts a tensor prepared by preprocessing elements such as qtimlvconverter and produces an output tensor that's parsed by postprocessing plugins such as qtimlvclassification, qtimlvdetection, qtimlvsegmentation, and qtimlvpose.

This plugin does the following:

- Accepts a tensor prepared by the preprocessing elements such as qtimlvconverter.
- Produces the output tensor that's parsed by the postprocessing plugins such as qtimlvclassification, qtimlvdetection, qtimlvsegmentation, and qtimlvpose.
- Uses the Qualcomm Neural Processing SDK library.
- Uses the ION/DMA buffers assigned by the GstMLBufferPool custom buffer pool class through IOCTL commands to the kernel.

The model and delegate usage is as follows:

- To use a model, provide the absolute path of the model through the model property.

  For some models, you may have to set the extra layers property.

After it's loaded, the model file provides the input and output capabilities:

- Number of tensors
- Dimensions
- Type

- If a delegate isn't specified, the plugin executes the model operations on the CPU.
- To select a different delegate, set the delegate property accordingly. For more information, see the Table 6-57.

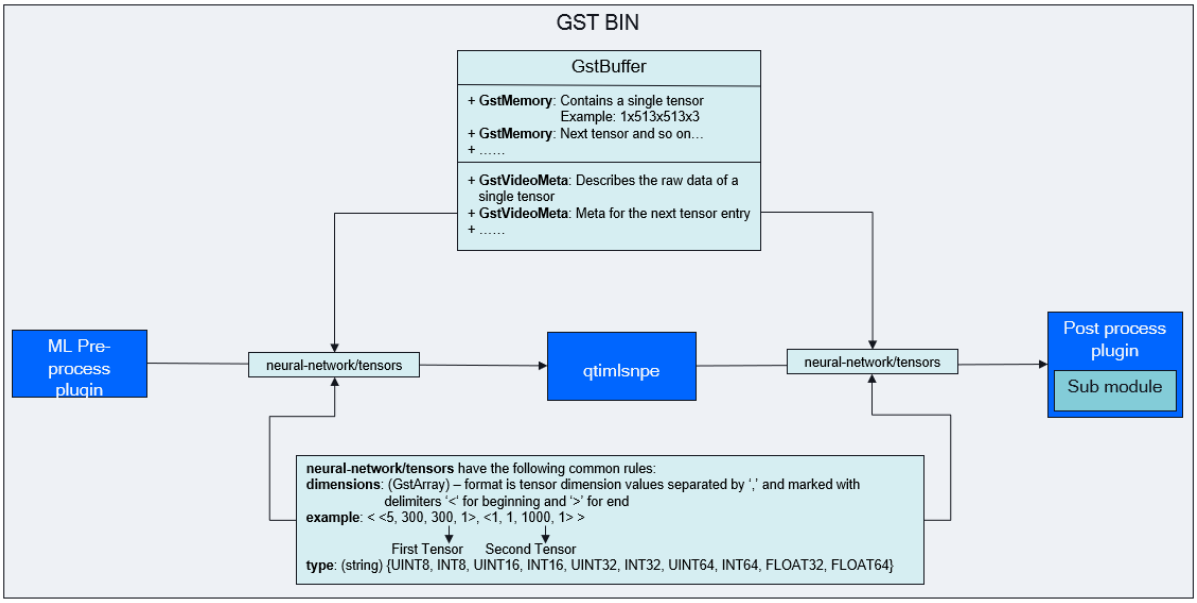


Figure 6-16 Tensor mode qtimlsnpe architecture

Inheritance chain

GObject → GstObject → GstElement → GstBaseTransform → GstMLSnpe

The following tables provide information on pad templates and element properties of qtimlsnpe. For use cases, see Qualcomm Neural Processing SDK use cases.

Pad configuration

Table 6-56 Pad templates for qtimlsnpe

Pad Name	Capabilities		
SINK template: 'sink' ▪ *Availability:* Always ▪ *Direction:* sink	neural-network/tensors	type:	{ (string)UINT8, (string)INT32, (string)FLOAT32 }
SRC template: 'src' ▪ *Availability:* Always ▪ *Direction:* source	neural-network/tensors	type:	{ (string)UINT8, (string)INT32, (string)FLOAT32 }

Element configuration

Table 6-57 Element properties of qtimlsnpe

Property	Description
name	The name of the object. ■ flags: readable, writable ■ String. Default: "mlsnpe0"
parent	The parent of the object. ■ flags: readable, writable ■ Object of type "GstObject"
model	Model filename. ■ flags: readable, writable ■ String. Default: null
delegate	Delegate the graph execution to another executor. ■ flags: readable, writable ■ Enum "GstMLSnpeDelegate" Default: 0, "none" ▫ (0): none - No delegate, CPU is used for all operations ▫ (1): dsp - Run the processing on the Hexagon DSP ▫ (2): gpu - Run the processing on the Adreno GPU ▫ (3): aip - Run the processing on Snapdragon AIX + HVX
layers	List of output layers. Should be set if the model has more than one output. ■ flags: readable, writable ■ GstValueArray of GValues of type "gchararray"

6.2.3.5 qtimltflite

The qtimltflite plugin shows the LiteRT capabilities (load and run the LiteRT models) as a GStreamer plugin.

This plugin does the following:

- Accepts a tensor prepared by the preprocessing elements such as qtimlvconverter.
- Produces the output tensor that's parsed by the postprocessing plugins such as qtimlvclassification, qtimlvdetection, qtimlvsegmentation, and qtimlvpose.
- Uses the LiteRT library.
- Uses the ION/DMA buffers allocated by the GstMLBufferPool custom buffer pool class through IOCTL commands to the kernel.

The model and delegate usage is as follows:

- To use a model, provide the absolute path of the model through the model property.

  After it's loaded, the model file provides the input and output capabilities:

  - Number of tensors
  - Dimensions
  - Type

- If a delegate isn't specified, the plugin executes the model operations on CPU.
- To select a different delegate, set the delegate property accordingly. For more information, see the Table 6-59.

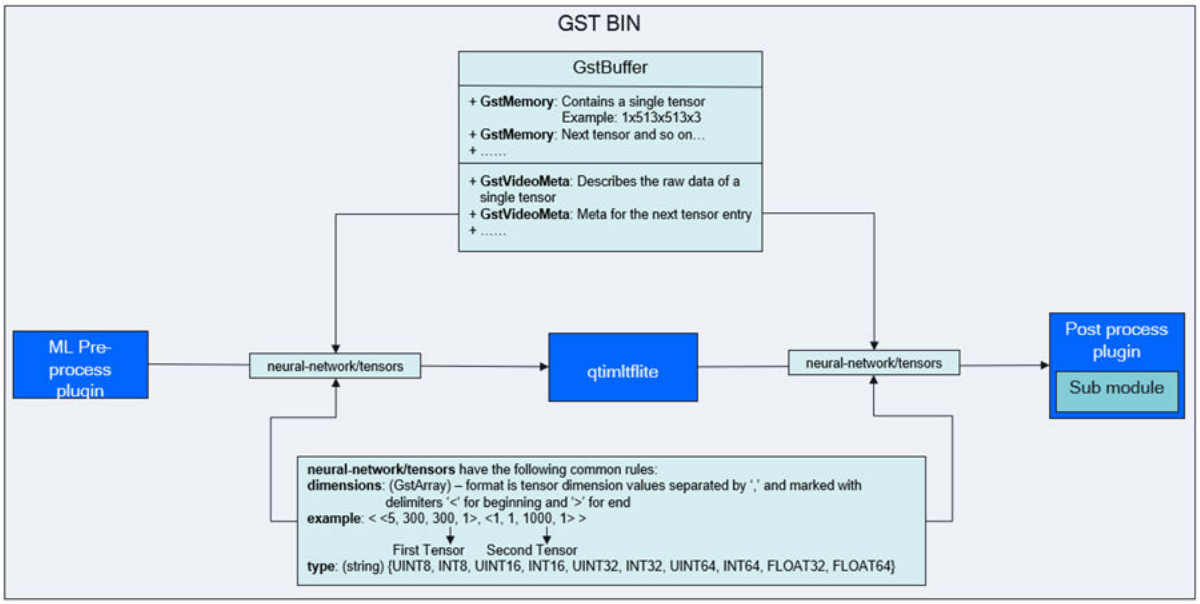


Figure 6-17 GstBuffer workflow with qtimltflite

Inheritance chain

GObject → GstObject → GstElement → GstBaseTransform → GstMLTFLite

The following tables provide information on pad templates and element properties of qtimltflite. For use cases, see LiteRT use cases.

Pad configuration

Table 6-58 Pad templates for qtimltflite

Pad Name	Capabilities		
SINK template: 'sink' ▪ *Availability:* Always ▪ *Direction:* sink	neural-network/tensors	type:	{ (string)UINT8, (string)INT32, (string)FLOAT32 }
SRC template: 'src' ▪ *Availability:* Always ▪ *Direction:* source	neural-network/tensors	type:	{ (string)UINT8, (string)INT32, (string)FLOAT32 }

Element configuration

Table 6-59 Element properties for qtimltflite

Property	Description
name	The name of the object.

Table 6-59 Element properties for qtimltflite (cont.)

Property	Description
	■ flags: readable, writable ■ String. Default: "mlTFLite0"
parent	The parent of the object. ■ flags: readable, writable ■ Object of type "GstObject"
model	Model filename. ■ flags: readable, writable ■ String. Default: null
delegate	Delegate part or all graph execution to another executor. ■ flags: readable, writable ■ Enum "GstMLTFLiteDelegate" Default: 0, "none" □ (0): none - No delegate, CPU is used for all operations □ (1): nnapi-dsp - Run the processing on the DSP through the NN API. Unsupported operations fall back on NPU, GPU or CPU □ (2): nnapi-gpu - Run the processing on the GPU through the NN API. Unsupported operations fall back on DSP, NPU or CPU □ (3): nnapi-npu - Run the processing on the NPU through the NN API. Unsupported operations fall back on DSP, GPU, or CPU □ (4): hexagon - Run the processing directly on the Hexagon DSP □ (5): gpu - Run the processing directly on the GPU □ (6): xnnpack - Run inferences using xnnpack cpu runtime
threads	Number of threads when running operations on the CPU delegate. ■ flags: readable, writable ■ Unsigned Integer. Range: 1 - 4 Default: 1

6.2.3.6 qtimlqnn

The qtimlqnn plugin shows the Qualcomm® AI Engine direct SDK capabilities (load and execute the Qualcomm Neural Network models) as a GStreamer plugin.

This plugin does the following:

- Accepts a tensor prepared by the preprocessing elements such as qtimlvconverter.
- Produces output tensor that's parsed by the postprocessing plugins such as qtimlvclassification, qtimlvdetection, qtimlvsegmentation, and qtimlvpose.
- Uses the Qualcomm Neural Network (QNN) library.
- Uses the ION/DMA buffers allocated by the GstMLBufferPool custom buffer pool class through IOCTL commands to the kernel.

The model and backend usage is as follows:

- To use a model, provide the absolute path of the model through the model property.

  After it's loaded, the model file provides the input and output capabilities:

  - Number of tensors
  - Dimensions
  - Type
- To select a different backend, set the backend property with the absolute path of the backend library.

Inheritance chain

GObject → GstObject → GstElement → GstBaseTransform→ GstMLQNN

The following tables provide information on pad templates and element properties of qtimlqnn.

Pad configuration

Table 6-60 Pad template for qtimlqnn

Pad Name	Capabilities		
SINK template: 'sink' ■ *Availability:* Always ■ *Direction:* sink	neural-network/ tensors	type:	{ (string)UINT8, (string)INT32, (string)FLOAT32 }
SRC template: 'src' ■ *Availability:* Always ■ *Direction:* source	neural-network/ tensors	type:	{ (string)UINT8, (string)INT32, (string)FLOAT32 }

Element configuration

Table 6-61 Element properties of qtimlqnn

Property	Description
name	The name of the object. ■ flags: readable, writable ■ String. Default: "mlqnn0"
parent	The parent of the object. ■ flags: readable, writable ■ Object of type "GstObject"
qos	Handle the quality-of-service events. ■ flags: readable, writable ■ Boolean. Default: false

Table 6-61 Element properties of qtimlqnn (cont.)

Property	Description
backend	the backend library path. ▪ flags: readable, writable ▪ String. Default: null
model	The model/cached binary file path. ▪ flags: readable, writable ▪ String. Default: null

6.2.3.7 qtimlvclassification

The qtimlvclassification plugin processes output tensors of an image classification model from the ML inference plugin (such as qtimltflite, qtimlsnpe, and qtimlqnn) into a result of predictions.

The negotiated GstCaps determines the processed output on the plug-in output. It can be either of the following:

- An image mask (GstCaps: video/x-raw), which is applied over the original image using qtivcomposer.
- A GStreamer formatted text (GstCaps: text/x-raw) containing the prediction results.

The qtimlvclassification plugin uses the CPU-based Cairo 2D graphics library for image overlay mask. The image overlay mask allows it to draw the prediction results in ION/DMA buffers. The GstImageBufferPool custom buffer pool class allocates the ION/DMA buffers through IOCTL commands to the kernel.

In the versatile text format, the prediction results are parsed into GStreamer-formatted string inside the buffers allocated using the regular system memory.

The module and labels properties of the plugin determine the method used for postprocessing operations.

- The module property specifies the postprocessing module, which runs dynamically at runtime with the available libraries at `in /usr/lib/gstreamer-1.0/ml/modules/` containing the prefix `ml-aclassification-`
- The labels property is a customized text file different for each machine learning detection model that you must provide for the prediction labels.

Optional properties are available for adjusting the prediction results.

- Use the `results` property to control the number of results that are displayed
- Use the `threshold` property to set a confidence threshold for prediction. The results with low confidence aren't displayed.

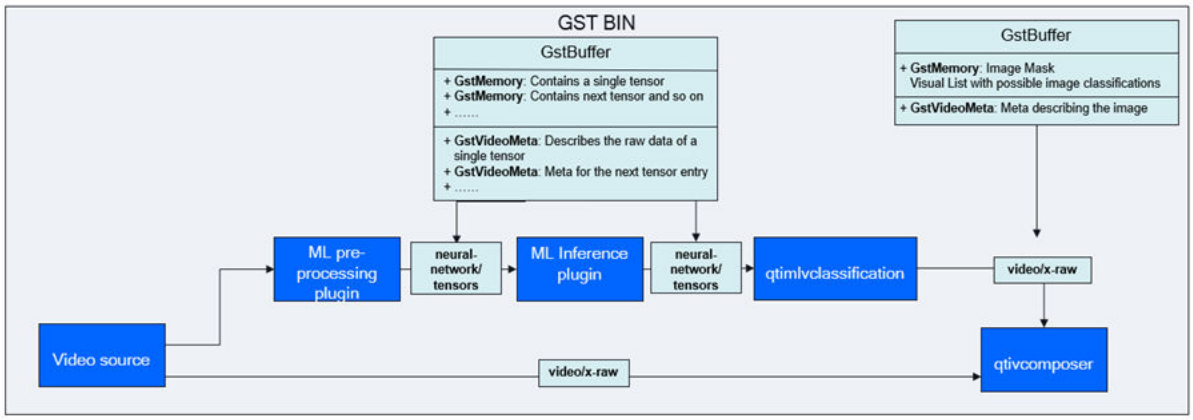


Figure 6-18 Postprocessing for object detection architecture

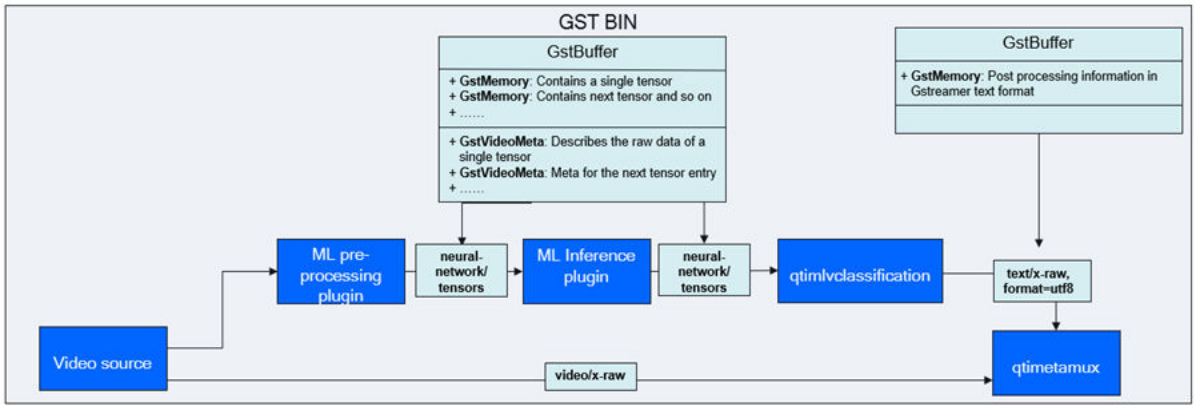


Figure 6-19 qtimlvclassification in pipeline

Inheritance chain

GObject → GstObject → GstElement → GstBaseTransform → GstMLVideoClassification

The following tables provide information on pad templates and element properties of qtimlsnpe. For use cases, see the classification use cases in LiteRT use cases and Qualcomm Neural Processing SDK use cases.

Pad configuration

Table 6-62 Pad templates for qtimlvclassification

Pad Name	Capabilities		
SINK template: 'sink' *Availability:* On request *Direction:* sink	neural-network/tensors	–	–
SRC template: 'src' *Availability:* Always *Direction:* source	video/x-raw	format:	{ (string)BGRA, (string)BGRx, (string)BGR16 }
	video/x-raw	format:	{ (string)BGRA, (string)BGRx, (string)BGR16 }
	text/x-raw	format:	{ (string)utf8 }

Element configuration

Table 6-63 Element properties of qtimlvclassification

Property	Description
name	The name of the object. ▪ flags: readable, writable ▪ String. Default: "mlvideoclassification0"
parent	The parent of the object. ▪ flags: readable, writable ▪ Object of type "GstObject"
qos	Handle the quality-of-service events. ▪ flags: readable, writable ▪ Boolean. Default: false
module	Module name that's used for processing the tensors. ▪ flags: readable, writable ▪ Enum "GstMLVideoClassificationModules" Default: 0, "none" ▫ (0): none - No module, default invalid mode ▫ (1): MobileNet - ml-vclassification-MobileNet
labels	The filename of the labels. ▪ flags: readable, writable ▪ String. Default: null
results	Number of results for display. ▪ flags: readable, writable ▪ Unsigned Integer. Range: 0 - 10 Default: 5
threshold	Confidence threshold in %. ▪ flags: readable, writable ▪ Double. Range: 10.0 - 100.0 Default: 10.0

6.2.3.8 qtimlaclassification

The qtimlaclassification plugin processes the output tensors of an audio classification model from the ML inference plugin ((such as qtimltflite) into a result of predictions.

The negotiated GstCaps determines the processed output on the plugin output.

For example, to infer the audio results from an MP4 file and integrate the audio classification with the original video frames, the output can be either of the following:

- An image mask (GstCaps: video/x-raw), which are applied over the original image using qtivcomposer.
- A GStreamer formatted text (GstCaps: text/x-raw) containing the prediction results.

The plugin uses the CPU-based Cairo 2D graphics library for image overlay mask. The image overlay mask enables Cairo to draw the prediction results in ION/DMA buffers. The GstImageBufferPool custom buffer pool class allocates the ION/DMA buffers through IOCTL commands to the kernel.

In the versatile text format, the prediction results are parsed into GStreamer-formatted string inside the buffers allocated using the regular system memory.

The module and labels properties of the plugin determine the method used for postprocessing operations.

- The module property specifies the postprocessing module, which runs dynamically at runtime with the available libraries at `in /usr/lib/gstreamer-1.0/ml/modules/` containing the prefix `ml-aclassification-`
- The labels property is a customized text file different for each machine learning detection model that you must provide for the prediction labels.

Optional properties are available for adjusting the prediction results.

- Use the `results` property to control the number of results that are displayed
- Use the `threshold` property to set a confidence threshold for prediction. The results with low confidence aren't displayed.

Inheritance chain

GObject → GstObject → GstElement → GstBaseTransform → GstMLAudioClassification

The following tables provide information on pad templates and element properties of qtimlsnpe. For use cases, Audio classification decode and display with LiteRT.

Pad configuration

Table 6-64 Pad templates for qtimlaclassification

Pad Name	Capabilities		
SINK template: 'sink' ▪ *Availability:* On request ▪ *Direction:* sink	neural-network/tensors	–	–
SRC template: 'src' ▪ *Availability:* Always ▪ *Direction:* source	video/x-raw	format:	{ (string)BGRA, (string)BGRx, (string)BGR16 }
	video/x-raw(memory:GBM)	format:	{ (string)BGRA, (string)BGRx, (string)BGR16 }
	text/x-raw	format:	{ (string)utf8 }

Element configuration

Table 6-65 Element properties of qtimlaclassification

Property	Description
name	The name of the object. ▪ flags: readable, writable ▪ String. Default: "mlaudioclassification0"
parent	The parent of the object. ▪ flags: readable, writable ▪ Object of type "GstObject"

Table 6-65 Element properties of qtimlaclassification (cont.)

Property	Description
qos	Handle the quality-of-service events. ▪ flags: readable, writable ▪ Boolean. Default: false
module	The module name that's used to process the tensors. ▪ flags: readable, writable ▪ Enum "GstMLAudioClassificationModules" Default: 0, "none" (0): none - No module, default invalid mode (1): yamnet - ml-aclassification-yamnet
labels	The labels filename. ▪ flags: readable, writable ▪ String. Default: null
results	The number of results to display. ▪ flags: readable, writable ▪ Unsigned Integer. Range: 0 - 10 Default: 5
threshold	The confidence threshold in %. ▪ flags: readable, writable ▪ Double. Range: 10.0 - 100.0 Default: 10.0

6.2.3.9 qtimlvdetection

The qtimlvdetection plugin processes output tensors of an object detection model from the ML inference plugin (such as qtimltflite, qtimlsnpe, and qtimlqnn) into result of predictions.

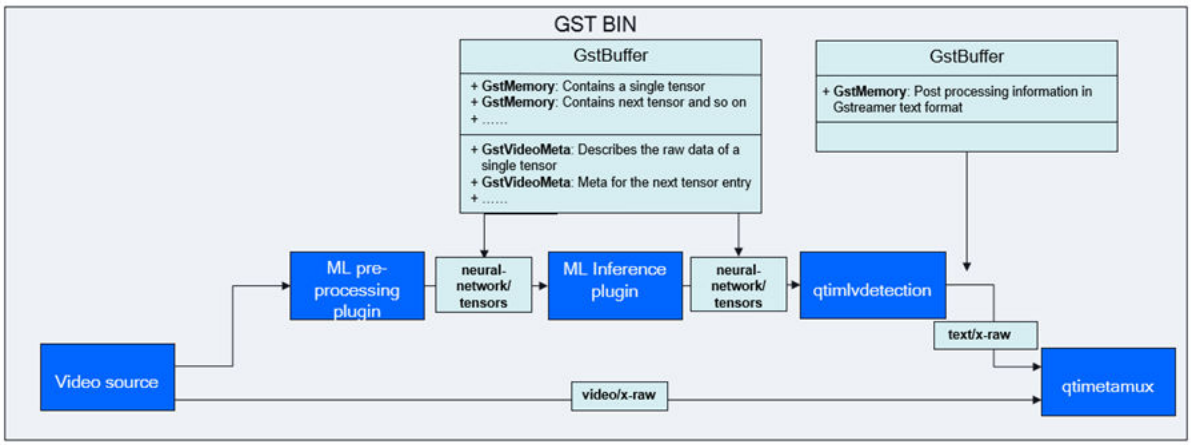


Figure 6-20 GstBuffer workflow with qtimlvdetection

Postprocessing

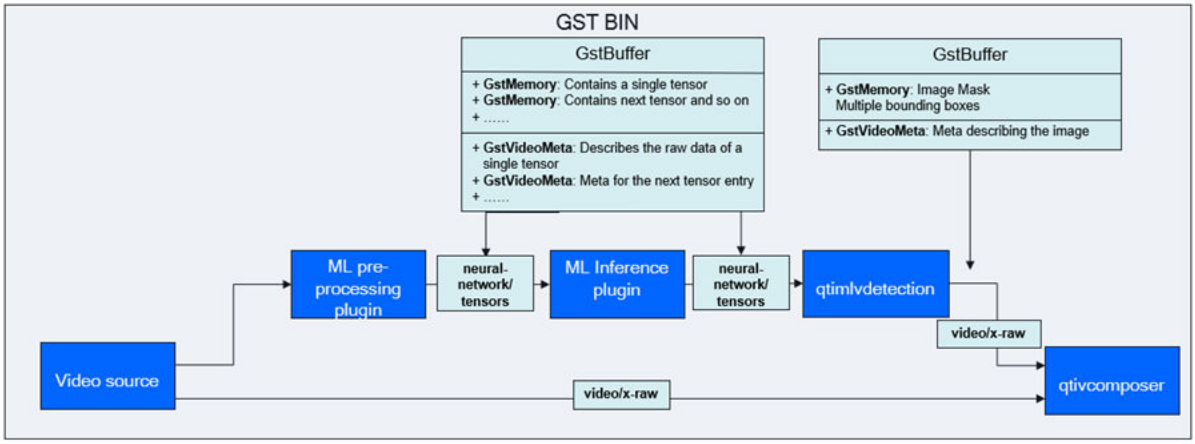


Figure 6-21 Postprocessing for object detection architecture

The negotiated GstCaps determines the processed output. The output can be either of the following:

- An image overlay mask (GstCaps: video/x-raw), which are applied over the original image using qtivcomposer.
  - The element uses the CPU-based Cairo 2D graphics library to draw the prediction results in ION/DMA buffers.
  - The GstImageBufferPool custom buffer pool class allocates the ION/DMA buffers through IOCTL commands to the kernel.
- The GStreamer-formatted text (GstCaps: text/x-raw) containing the prediction results.

  The prediction results are parsed into GStreamer-formatted string inside buffers allocated using regular system memory.

The module and labels properties of the plugin determine the method for the postprocessing operations.

- The module property specifies the postprocessing module. It's populated dynamically at runtime with the libraries available in `/usr/lib/gstreamer-1.0/ml/modules/` containing the prefix "`ml-vdetection-`".
- The labels property is a customized text file different for each machine learning detection model that you must provide for the prediction labels.

Optional properties are available for adjusting the prediction results.

- Use the results property to control the number of results that are displayed
- Use the threshold property to set a confidence threshold for prediction. The results with low confidence aren't displayed.

Inheritance chain

GObject → GstObject → GstElement → GstBaseTransform → GstMLVideoDetection

The following tables provide information on pad templates and element properties of qtimlvdetection. For use cases, see the detection use cases in LiteRT use cases and Qualcomm Neural Processing SDK use cases.

Pad configuration

Table 6-66 Pad templates for qtimlvdetection

Pad Name	Capabilities		
SINK template: 'sink' ■ *Availability:* On request ■ *Direction:* sink	neural-network/tensors	–	–
SRC template: 'src' ■ *Availability:* Always ■ *Direction:* source	video/x-raw	format:	{ (string)BGRA, (string)BGRx, (string)BGR16 }
	video/x-raw	format:	{ (string)BGRA, (string)BGRx, (string)BGR16 }
	text/x-raw	format:	{ (string)utf8 }

Element configuration

Table 6-67 Element properties for qtimlvdetection

Property	Description
name	The name of the object. ■ flags: readable, writable ■ String. Default: "mlvideodetection0"
parent	The parent of the object. ■ flags: readable, writable ■ Object of type "GstObject"
qos	Handle the quality-of-service events. ■ flags: readable, writable ■ Boolean. Default: false
module	Module name that's used for processing the tensors. ■ flags: readable, writable ■ Enum "GstMLVideoDetectionModules" (0): none - No module, default invalid mode ▫ (1): face-detect - ml-vdetection-face-detect ▫ (2): ssd-mobilenet - ml-vdetection-ssd-MobileNet ▫ (3): YOLOv5m - ml-vdetection-yolov5m ▫ (4): YOLOv5s - ml-vdetection-yolov5s
labels	The filename of the labels. ■ flags: readable, writable ■ String. Default: null
results	Number of results to display. ■ flags: readable, writable ■ Unsigned Integer. Range: 0 - 10 Default: 5
threshold	Confidence threshold in %. ■ flags: readable, writable ■ Double Range: 10.0 - 100.0 Default: 10.0

6.2.3.10 qtimlvsegmentation

The qtimlvsegmentation plugin processes output tensors of an image segmentation/depth estimation model from the ML inference plugin (such as qtimltflite, qtimlsnpe, and qtimlqnn) into result of predictions.

The processed output is an image mask (GstCaps: video/x-raw). GstCaps determines the dimensions and format, which is applied over the original image using qtivcomposer.

For this mask, the element uses the CPU based Cairo 2D graphics library to draw the prediction results in ION/DMA buffers. The GstImageBufferPool custom buffer pool class allocates the ION/DMA buffers through IOCTL commands to the kernel.

The module and labels properties of the plugin determine the method for the postprocessing operations.

- The module property specifies the postprocessing module. It's populated dynamically at runtime with the libraries available in `/usr/lib/gstreamer-1.0/ml/modules/` containing the prefix "`ml-vsegmentation--`".
- The labels property is a customized text file different for each machine learning detection model that you must provide for the prediction labels.

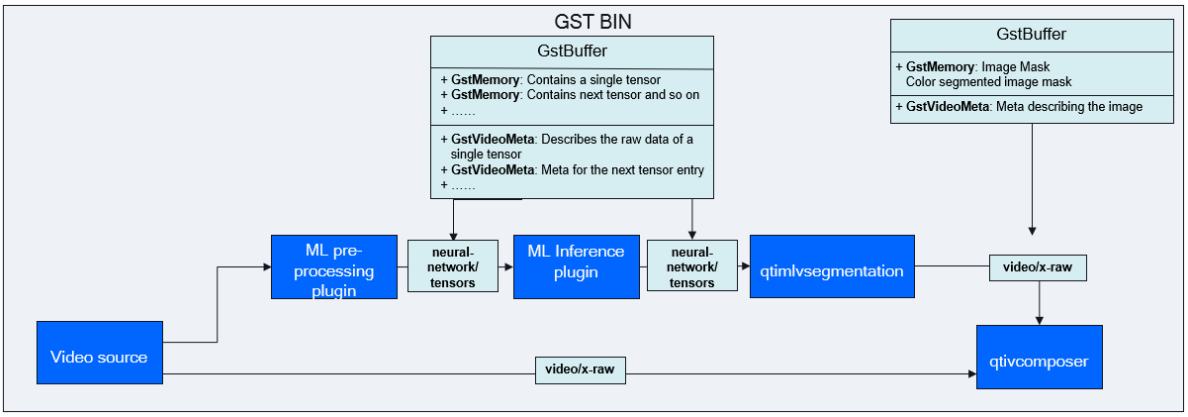


Figure 6-22 GstBuffer workflow with qtimlvsegmentation

Inheritance chain

GObject → GstObject → GstElement → GstBaseTransform → GstMLVideoSegmentation

The following tables provide information on pad templates and element properties of qtimlvsegmentation. For use cases, see the segmentation use cases in LiteRT use cases and Qualcomm Neural Processing SDK use cases.

Pad configuration

Table 6-68 Pad templates for qtimlvsegmentation

Pad Name	Capabilities		
SINK template: 'sink' ■ *Availability:* On request ■ *Direction:* sink	neural-network/tensors	–	–
SRC template: 'src' ■ *Availability:* Always ■ *Direction:* source	video/x-raw	format:	{ (string)BGRA, (string)BGRx, (string)BGR16 }
	text/x-raw	format:	{ (string)utf8 }

Element configuration

Table 6-69 Element properties for qtimlvsegmentation

Property	Description
name	The name of the object. ■ flags: readable, writable ■ String. Default: "mlvideosegmentation0"
parent	The parent of the object. ■ flags: readable, writable ■ Object of type "GstObject"
qos	Handle Quality-of-Service events. ■ flags: readable, writable ■ Boolean. Default: false
module	Module name that's used for processing the tensors. ■ flags: readable, writable ■ Enum "GstMLVideoSegmentationModules" Default: 0, "none" ▫ (0): none - No module, default invalid mode ▫ (1): deeplab-argmax - ml-vsegmentation-deeplab-argmax ▫ (2): midas-v2 - ml-vsegmentation-midas-v2
labels	The filename of the labels. ■ flags: readable, writable ■ String. Default: null

6.2.3.11 qtimlvpose

The qtimlvpose plugin processes output tensors of a pose estimation model from the ML inference plugin (such as qtimltflite, qtimlsnpe, and qtimlqnn) into result of predictions.

Postprocessing

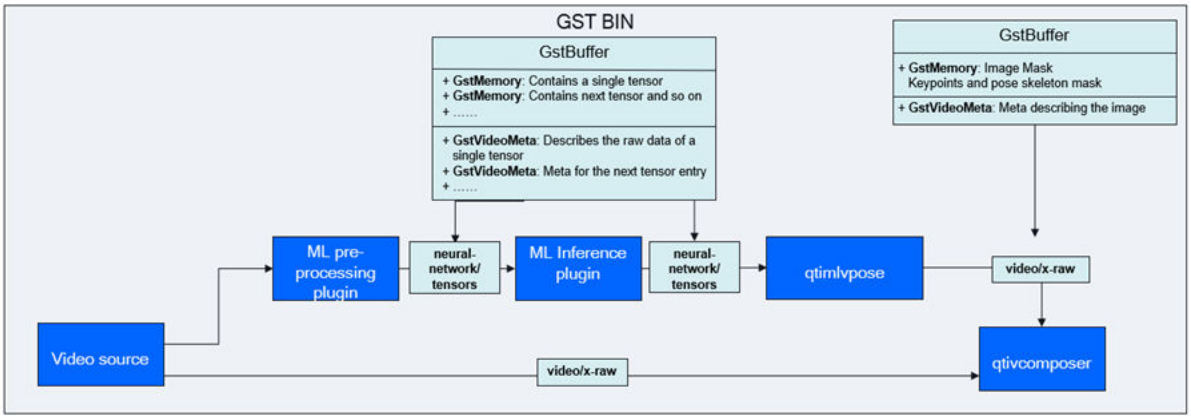


Figure 6-23 Postprocessing for pose estimation architecture

The negotiated GstCaps on the plugin output determines the processed output, which can be either of the following:

- An image mask (GstCaps: video/x-raw) that's applied over the original image using qtivcomposer.
  - The element uses the CPU-based Cairo 2D graphics library to draw the prediction results in ION/DMA buffers.
  - The GstImageBufferPool custom buffer pool class allocates the ION/DMA buffers through IOCTL commands to the kernel.
- A GStreamer-formatted text (GstCaps: text/x-raw) containing the prediction results.

  The prediction results are parsed into GStreamer-formatted string inside buffers allocated using regular system memory.

The module and labels properties of the plugin determine the method for the postprocessing operations.

- The module property specifies the postprocessing module and is populated dynamically at run time with the libraries available in `/usr/lib/gstreamer-1.0/ml/modules/` containing the prefix "`ml-vpose-`".
- The labels property is a customized text file different for each machine learning detection model that you must provide for the prediction labels.

Optional properties are available for adjusting the prediction results.

- Use the results property to control the number of results that are displayed
- Use the threshold property to set a confidence threshold for prediction. The results with low confidence aren't displayed.

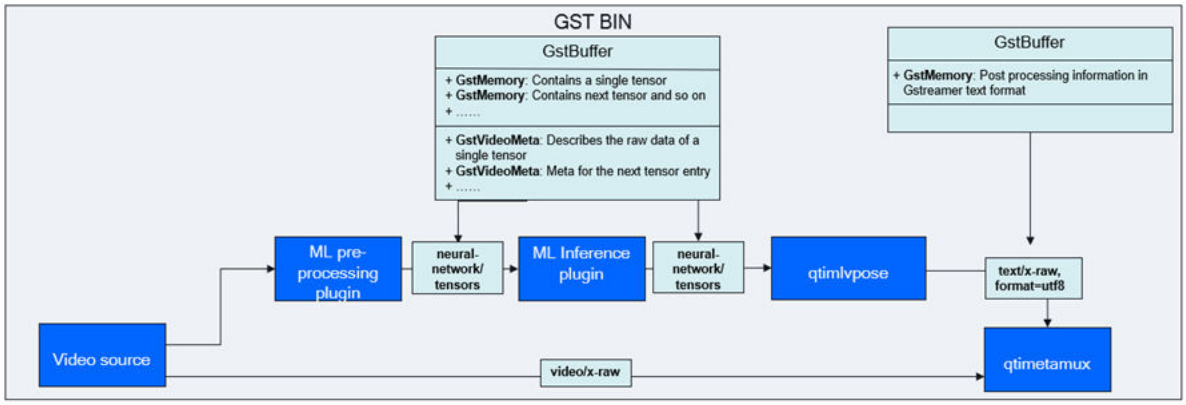


Figure 6-24 GstBuffer workflow with qtimlvpose

Inheritance chain

GObject → GstObject → GstElement → GstBaseTransform → GstMLVideoPose

The following tables provide information on pad templates and element properties of qtimlvpose. For use cases, see the pose estimation use cases in LiteRT use cases.

Pad configuration

Table 6-70 Pad templates for qtimlvpose

Pad Name	Capabilities		
SINK template: 'sink' *Availability:* On request *Direction:* sink	neural-network/tensors	–	–
SRC template: 'src' *Availability:* Always *Direction:* source	video/x-raw	format:	{ (string)BGRA, (string)BGRx, (string)BGR16 }
	text/x-raw	format:	{ (string)utf8 }

Element configuration

Table 6-71 Element properties of qtimlvpose

Property	Description
name	The name of the object. ▪ flags: readable, writable ▪ String. Default: "mlvideopose0"
parent	The parent of the object. ▪ flags: readable, writable ▪ Object of type "GstObject"
qos	Handle Quality-of-Service events. ▪ flags: readable, writable ▪ Boolean. Default: false

Table 6-71 Element properties of qtimlvpose (cont.)

Property	Description
module	Module name that's going to be used for processing the tensors. ▪ flags: readable, writable ▪ Enum "GstMLVideoPoseModules" Default: 0, "none" ▫ (0): none - No module, default invalid mode ▫ (1): posenet - ml-vpose-posenet
labels	The file name of the labels. ▪ flags: readable, writable ▪ String. Default: null
results	Number of results to display. ▪ flags: readable, writable ▪ Unsigned Integer. Range: 0 - 10 Default: 5
threshold	Confidence threshold in %. ▪ flags: readable, writable ▪ Double. Range: 10.0 - 100.0 Default: 50.0

6.2.3.12 qtimlvsuperresolution

The qtimlvsuperresolution plugin processes output tensors of an image super resolution model from the ML inference plugin (such as qtimltflite or qtimlsnpe).

The processed output is an upscaled image (GstCaps: video/x-raw). GstCaps determines the dimensions and format, which is applied over the original image using qtivcomposer.

The module property of the plugin determines the method for the postprocessing operations.

- The property specifies the postprocessing module.
- The property is populated dynamically at runtime with the libraries available in `/usr/lib/gstreamer-1.0/ml/modules/` containing the prefix `libml-vsuperresolution-`.

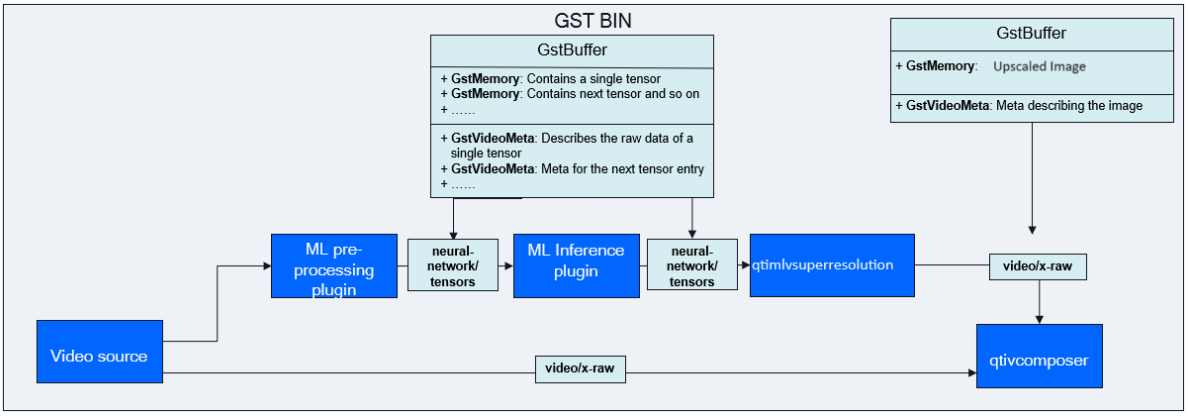


Figure 6-25 GstBuffer workflow for qtimlvsuperresolution

Inheritance chain

GObject → GstObject → GstElement → GstBaseTransform → GstMLVSuperResolution

The following tables provide information on pad templates and element properties of qtimlvsuperresolution. For usage, see Video super resolution and display with LiteRT.

Pad configuration

Table 6-72 Pad templates for qtimlvsuperresolution

Pad Name	Capabilities		
SINK template: 'sink' ▪ *Availability:* On request ▪ *Direction:* sink	neural-network/tensors	–	–
SRC template: 'src' ▪ *Availability:* Always ▪ *Direction:* source	video/x-raw	format:	{ (string)BGRA, (string)BGRx, (string)BGR16 }

Element configuration

Table 6-73 Element properties for qtimlvsuperresolution

Property	Description
name	The name of the object. ▪ flags: readable, writable ▪ String. Default: "mlvsuperresolution0"
parent	The parent of the object. ▪ flags: readable, writable ▪ Object of type "GstObject"
qos	Handle Quality-of-Service events. ▪ flags: readable, writable ▪ Boolean. Default: false
module	Module name to be used for processing the tensors. ▪ flags: readable, writable ▪ Enum "GstMLVideoSuperResolutionModules" Default: 0, "none" (0): none - No module, default invalid mode (1): srnet - ml-vsuperresolution-srnet
labels	The file name for labels. ▪ flags: readable, writable ▪ String. Default: null

RELATED INFORMATION

"Video super-resolution" on page 71

6.2.3.13 qtivoverlay

The qtivoverlay plugin is a hardware accelerated in-place image draw and blit plugin for drawing overlays on top of the YUV images.

You can configure the overlays through properties and through the buffer metadata attached to the input buffers.

- It supports blitting of static images, bounding boxes, custom user text, data/time overlays, and privacy mask through properties.
- It supports blitting of detection (bounding box), segmentation (semantic mask/mask image), classification (label/user text), and pose graph overlay through metadata attached to the input buffer.

Inheritance chain

GObject → GstObject → GstElement → GstBaseTransform → GstVideoFliter → GstVideoOverlay

The following tables provide information on pad templates and element properties of qtivoverlay. For use cases, see Video overlay use cases.

Pad configuration

Table 6-74 Pad templates for qtivoverlay

Pad Name	Capabilities		
SINK template: 'sink' ▪ *Availability:* Always ▪ *Direction:* sink	video/x-raw	format: width: height: framerate:	{ (string)NV12, (string)NV21 } [1, 32767] [1, 32767] [0/1, 2147483647/1]
	video/x-raw (ANY)	format: width: height: framerate:	{ (string)NV12, (string)NV21 } [1, 32767] [1, 32767] [0/1, 2147483647/1]
SRC template: 'src' ▪ *Availability:* Always ▪ *Direction:* source	video/x-raw	format: width: height: framerate:	{ (string)NV12, (string)NV21 } [1, 32767] [1, 32767] [0/1, 2147483647/1]
	video/x-raw (ANY)	format: width: height: framerate:	{ (string)NV12, (string)NV21 } [1, 32767] [1, 32767] [0/1, 2147483647/1]

Element configuration

Table 6-75 Element properties for qtivoverlay

Property	Description
name	The name of the object. ▪ flags: readable, writable ▪ String. Default: "overlay0"
parent	The parent of the object. ▪ flags: readable, writable ▪ Object of type "GstObject"
qos	Handles the quality-of-service (QoS) events. ▪ flags: readable, writable ▪ Boolean. Default: true
overlay-text	Renders text on top of video stream. ▪ flags: readable, writable, changeable in NULL, READY, PAUSED or PLAYING state ▪ String. Default: ""
overlay-date	Renders date and time on top of video stream. ▪ flags: readable, writable, changeable in NULL, READY, PAUSED or PLAYING state ▪ String. Default: ""
overlay-simg	Renders static image on top of video stream. ▪ flags: readable, writable, changeable in NULL, READY, PAUSED or PLAYING state ▪ String. Default: ""
overlay-bbox	Renders bounding box and label on top of video stream. ▪ flags: readable, writable, changeable in NULL, READY, PAUSED or PLAYING state ▪ String. Default: ""
overlay-mask	Renders privacy mask on top of video stream. ▪ flags: readable, writable, changeable in NULL, READY, PAUSED or PLAYING state ▪ String. Default: ""
bbox-color	Bounding box overlay color. ▪ flags: readable, writable ▪ Unsigned Integer. Range: 0 - 4294967295 Default: 52479
date-color	Date overlay color. ▪ flags: readable, writable ▪ Unsigned Integer. Range: 0 - 4294967295 Default: 4278190335
text-color	Text overlay color. ▪ flags: readable, writable ▪ Unsigned Integer. Range: 0 - 4294967295 Default: 4294902015
pose-color	Pose overlay color. ▪ flags: readable, writable ▪ Unsigned Integer. Range: 0 - 4294967295 Default: 869007615
arrows-color	Arrows overlay color.

Table 6-75 Element properties for qtivoverlay (cont.)

Property	Description
	▪ flags: readable, writable ▪ Unsigned Integer. Range: 0 - 4294967295 Default: 4278190335
bbox-font-size	Bounding box overlay font size. ▪ flags: readable, writable ▪ Unsigned Integer. Range: 1 - 4294967295 Default: 25
date-font-size	Date overlay font size. ▪ flags: readable, writable ▪ Unsigned Integer. Range: 1 - 4294967295 Default: 20
text-font-size	Text overlay font size. ▪ flags: readable, writable ▪ Unsigned Integer. Range: 1 - 4294967295 Default: 40
dest-rect-ml-text	Destination rectangle parameters for ML detection overlay. The Start-X, Start-Y , Width, and Height of the destination rectangle format is <X, Y, WIDTH, HEIGHT>. ▪ flags: readable, writable ▪ GstValueArray of GValues of type "gint"
Polygon	
arrows-ft-mv	Arrows MV filter. ▪ flags: readable, writable ▪ Unsigned Integer. Range: 0 - 4294967295 Default: 0
arrows-ft-sad	Arrows sad filter. ▪ flags: readable, writable ▪ Unsigned Integer. Range: 0 - 4294967295 Default: 0
arrows-ft-var	Arrows var filter. ▪ flags: readable, writable ▪ Unsigned Integer. Range: 0 - 4294967295 Default: 0

6.2.3.14 qtimetamux

The qtimetamux plugin uses frame matching techniques to associate or attach ML string based postprocessing results (output from the postprocessing plugin) or CV information to original frame such as GstMeta.

This output is shared with plugins such as qtioverlay for rendering or passed down the ML inference pipeline for further inferencing.

It's also possible to attach an appsink plugin and receive the GstBuffer along with the GstMeta in the user-defined application.

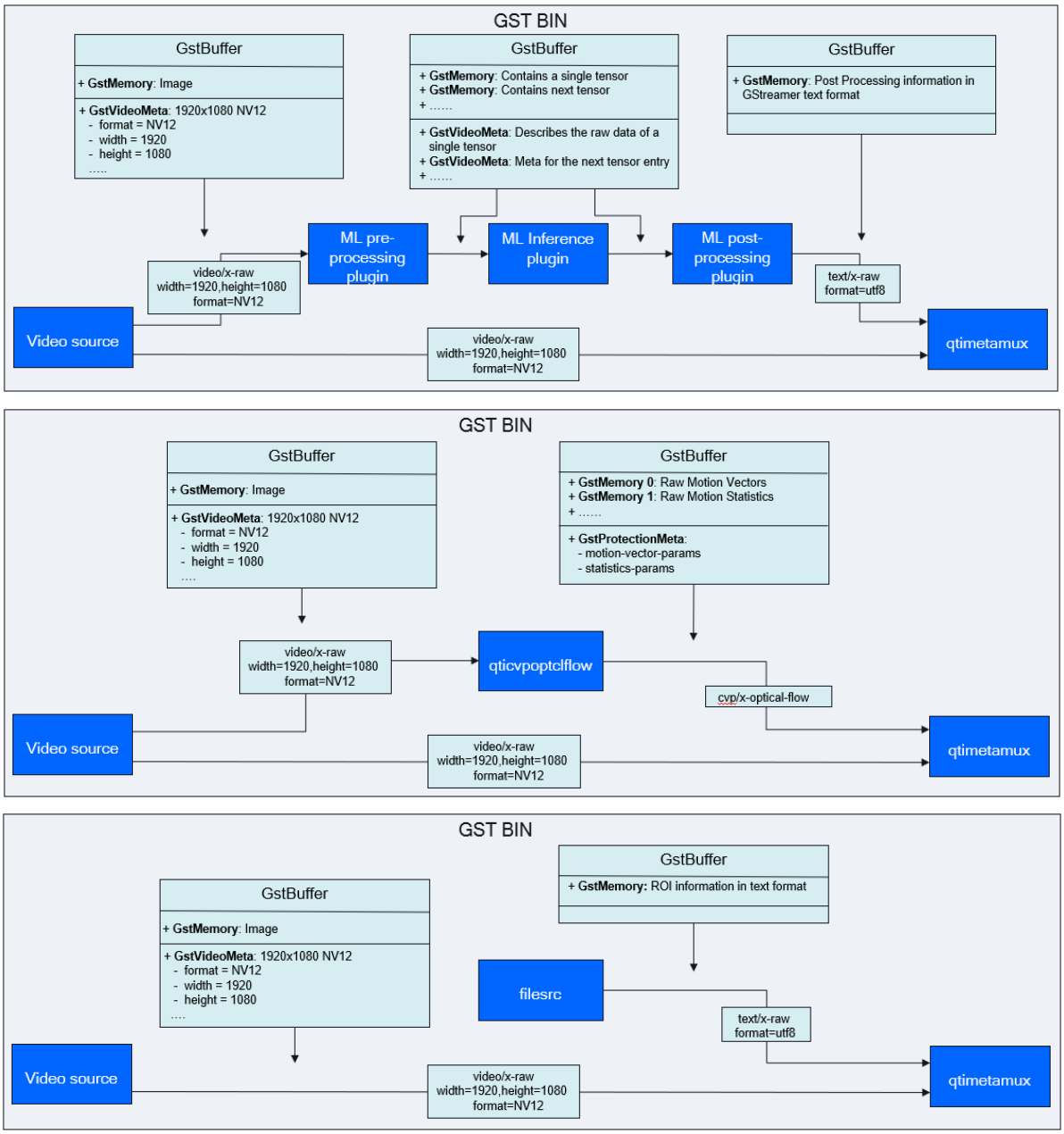


Figure 6-26 GStreamer pipeline with qtimetamux

Inheritance chain

GObject → GstObject → GstElement → GstMetaMux

The following tables provide information on pad templates and element properties of qtimetamux. For use cases, see LiteRT use cases and Daisy chain detection and classification.

Pad configuration

Table 6-76 Pad templates for qtimetamux

Pad Name	Capabilities		
SINK template: 'sink' ■ *Availability:* Always ■ *Direction:* sink	video/x-raw (ANY)	–	–
	video/x-raw	–	–
SINK template: 'data_%u' ■ *Availability:* On request ■ *Direction:* sink	text/x-raw	format:	utf8
SRC template: 'src' ■ *Availability:* Always ■ *Direction:* source	video/x-raw (ANY)	–	–
	video/x-raw	–	–

Element configuration

Table 6-77 Element properties of qtimetamux

Property	Description
name	The name of the object. ■ flags: readable, writable ■ String. Default: "metamux0"
parent	The parent of the object. ■ flags: readable, writable ■ Object of type "GstObject"

6.2.3.15 qtimldemux

The qtimldemux element splits batched (such as first tensor dimension is greater and 1) tensors (GstMemory blocks) from a single input GstBuffer into separate GstBuffers containing an unbatched tensors (GstMemory blocks).

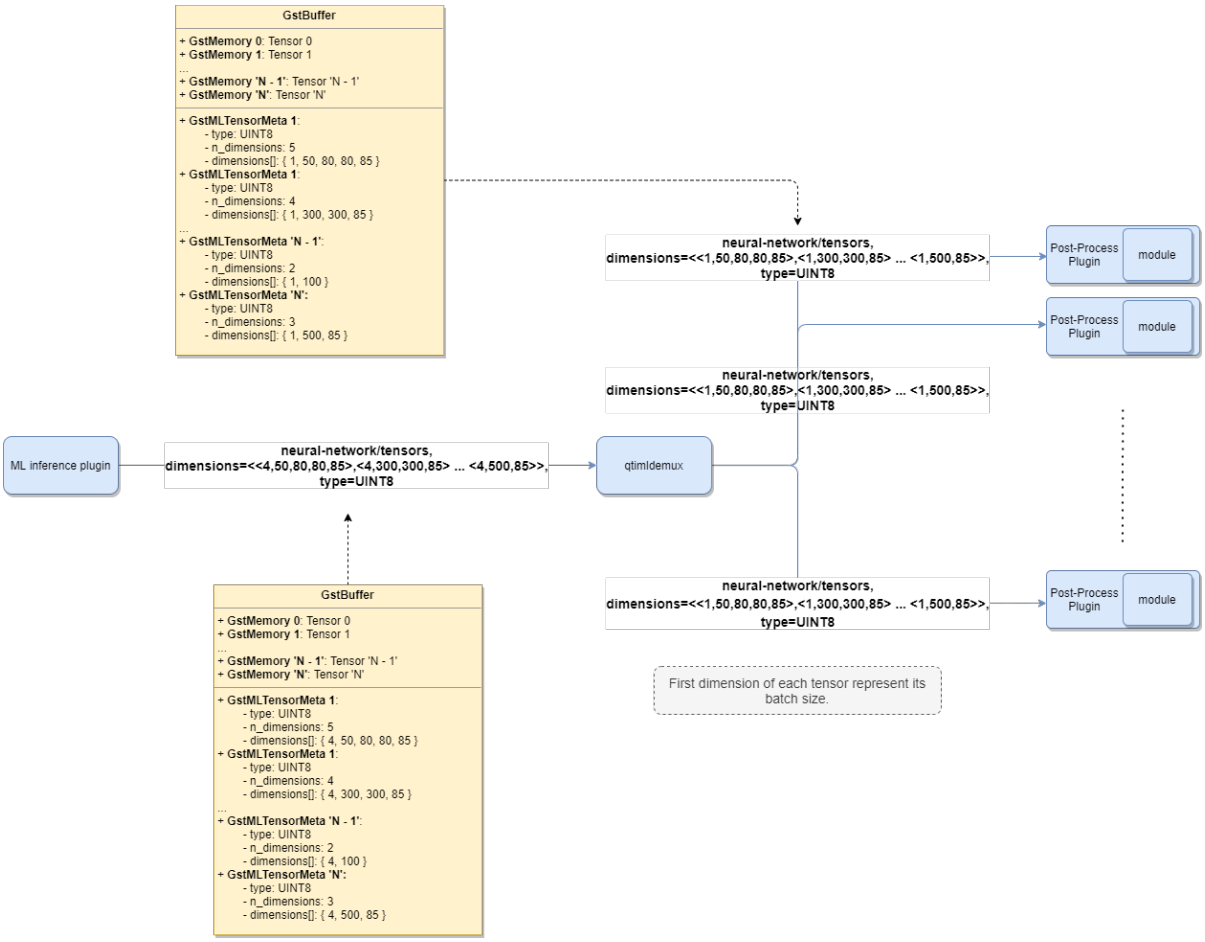


Figure 6-27 qtimldemux architecture

Inheritance chain

GObject → GstObject → GstElement → GstMLDemux

The following tables provide information on pad templates and element properties of qtimldemux. For use cases, see Image classification.

Pad configuration

Table 6-78 Pad template for qtimldemux

Pad Name	Capabilities		
SINK template: 'sink' ▪ *Availability:* On request ▪ *Direction:* sink	neural-network/ tensors	type:	{ (string)INT8, (string)UINT8, (string)INT32, (string)UINT32, (string)FLOAT16, (string)FLOAT32 }
SRC template: 'src_%u' ▪ *Availability:* Always ▪ *Direction:* source	neural-network/ tensors	type:	{ (string)INT8, (string)UINT8, (string)INT32, (string)UINT32, (string)FLOAT16, (string)FLOAT32 }

Element configuration

Table 6-79 qtimldemux element properties

Property	Description
name	The name of the object. ▪ flags: readable, writable ▪ String. Default: "mldemux0"
parent	The parent of the object. ▪ flags: readable, writable ▪ Object of type "GstObject"

6.2.3.16 qtirtspbin

The qtirtspbin plugin transmits the video or data streams using RTSP.

The plugin can transmit the encoded video, text data (ML data), or serialized data (x-rtp packet).

When the plugin starts, it opens a default port such as `127.0.0.1:8900` where many clients (media players such as VLC) can connect.

Inheritance chain

GObject → GstObject → GstElement → GstBin → GstRtspBin

The following tables provide information on pad templates and element properties of qtirtspbin. For use cases, see Single stream from camera to RTSP with ML detection.

Pad configuration

Table 6-80 Pad templates for qtirtspbin

Pad Name	Capabilities
▪ SINK template: 'sink_%u' ▪ Availability: On request	▪ video/x-h264 ▪ video/x-h265 ▪ audio/mpeg

Table 6-80 Pad templates for qtirtspbin (cont.)

Pad Name	Capabilities
	▪ text/x-raw ▪ application/x-rtp

Element configuration

Table 6-81 Element properties of qtirtspbin

Property	Description
name	The name of the object. ▪ flags: readable, writable ▪ String. Default: "rtspbin0"
parent	The parent of the object. ▪ flags: readable, writable ▪ Object of type "GstObject"
address	The IP address of the server. ▪ flags: readable, writable ▪ String. Default: "127.0.0.1"
port	The port for listening. ▪ flags: readable, writable ▪ String. Default: "8900"
mpoint	The mounting point. ▪ flags: readable, writable ▪ String. Default: "/live"
async-handling	The bin that handles the asynchronous state changes. ▪ flags: readable, writable ▪ Boolean. Default: false
message-forward	Forwards all the children messages. ▪ flags: readable, writable ▪ Boolean. Default: false

6.2.3.17 qtismartvencbin

The qtismartvencbin plugin uses machine learning and computer vision to reduce the video bandwidth. It can be used for surveillance, where only the small image areas are in motion. The bitrate reduction can be configured according to the quality requirements.

The qtismartvencbin plugin implements the following:

- A GStreamer bin that has a video encoder and integrates computer vision APIs.
- The post-processed person-detection results from the machine learning pipeline.

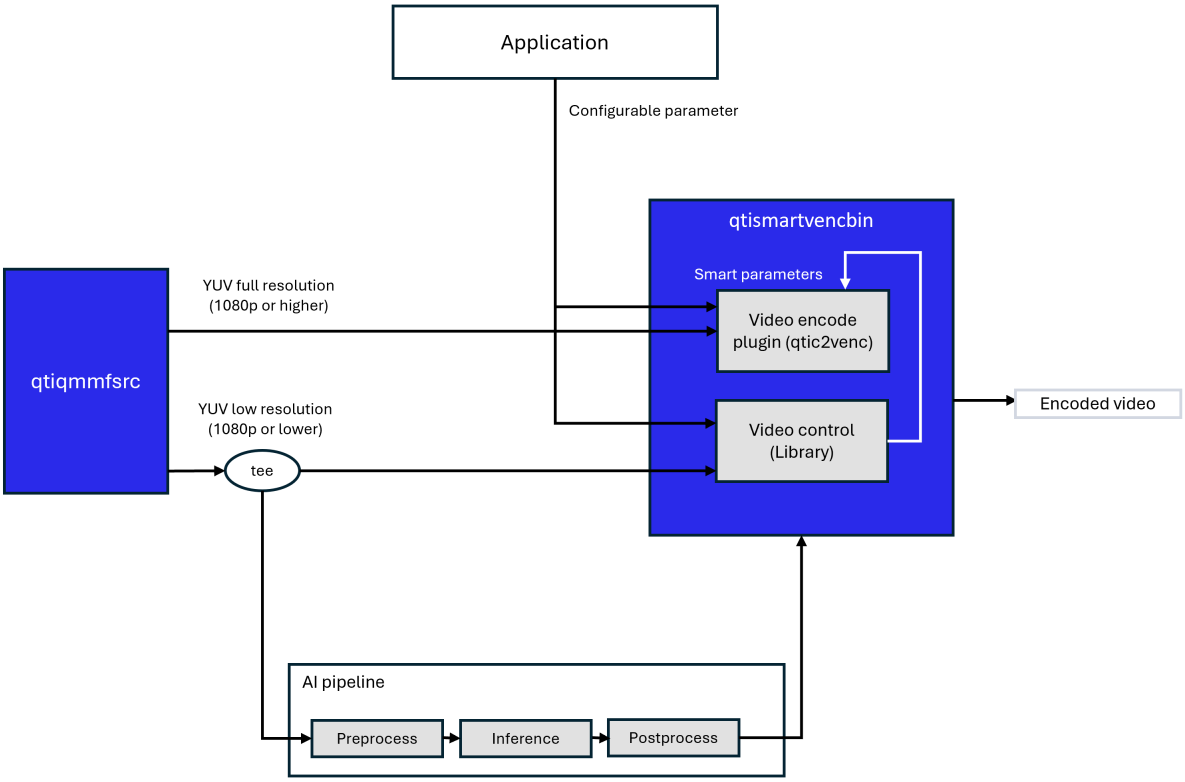


Figure 6-28 qtismartvencbin architecture

The following tables provide information on pad templates and element properties of qtismartvencbin. For use cases, see Smart codec.

Pad configuration

Table 6-82 Pad template for qtismartvencbin

Pad Name	Capabilities		
SINK template: 'sink' ▪ *Availability:* Always ▪ *Direction:* sink	video/x-raw	format	{NV12}
		width	[1, 2147483647]
		height	[1, 2147483647]
		height	[0/1, 2147483647/1]
	video/x-raw(memory:GBM)	format	{NV12}
		width	[1, 2147483647]
		height	[1, 2147483647]
		height	[0/1, 2147483647/1]
SINK template: 'sink_ctrl' ▪ *Availability:* Always ▪ *Direction:* sink	video/x-raw	format	{NV12}
		width	[1, 2147483647]
		height	[1, 2147483647]
		framerate	[0/1, 2147483647/1]

Table 6-82 Pad template for qtismartvencbin (cont.)

Pad Name	Capabilities		
	video/x-raw(memory:GBM)	format	{NV12}
		width	[1, 2147483647]
		height	[1, 2147483647]
		framerate	[0/1, 2147483647/1]
SINK template: 'sink_ml' ■ *Availability:* Always ■ *Direction:* sink	text/x-raw	format	{utf-8}
SRC template: 'src' ■ *Availability:* Always ■ *Direction:* source	video/x-h264 video/x-h265	–	–

Element configuration

Table 6-83 Element properties of qtismartvencbin

Property	Description
async-handling	The bin handles the asynchronous state changes. ■ flags: readable, writable ■ Boolean. Default: false
default-gop	The default GOP length. ■ flags: readable, writable, changeable only in NULL or READY state ■ Unsigned Integer. Range: 0 - 4294967295 Default: 30
encoder	The encoder to use (callable only in NULL state). ■ flags: readable, writable, changeable only in NULL or READY state ■ Enum "GstBinEncoderType" Default: 0, "c2enc" (0): c2enc - Codec2 encoder. (1): omxenc - OMX encoder. (2): v4l2h264enc - V4L2 H264 encoder (3): v4l2h265enc - V4L2 H265 encoder
levels-override	The override framerate levels. For example: `"LevelsOverride,fr_static=16,fr_low=4,fr_medium=2,fr_high=1;"` ■ flags: readable, writable ■ string. Default: null
max-bitrate	The maximum bitrate in bits per second. ■ flags: readable, writable, changeable only in NULL or READY state ■ Unsigned Integer. Range: 0 - 4294967295 Default: 6000000
max-gop	The maximum GOP length. ■ flags: readable, writable, changeable only in NULL or READY state ■ Unsigned Integer. Range: 0 - 4294967295 Default: 600

Table 6-83 Element properties of qtismartvencbin (cont.)

Property	Description
message-forward	Forwards all children messages. ▪ flags: readable, writable ▪ Boolean. Default: false
name	The name of the object. ▪ flags: readable, writable, 0x2000 ▪ String. Default: "videoencbin0"
parent	The parent of the object. ▪ flags: readable, writable, 0x2000 ▪ Object of type "GstObject"
roi-quality-cfg	The ROI quality configuration. For example:`"ROIQPs,car=2,person=1,tree=-2;"` ▪ flags: readable, writable ▪ String. Default: null
smart-framerate	Enable/disable the smart framerate functionality. ▪ flags: readable, writable, changeable only in NULL or READY state ▪ Boolean. Default: true
smart-gop	Enable/disable the smart GOP functionality. ▪ flags: readable, writable, changeable only in NULL or READY state ▪ Boolean. Default: true

6.3 Develop plugins using Qualcomm IM SDK

The GStreamer plugins allow you to effectively use and implement the Qualcomm IM SDK functionality. Download and install eSDK to develop or customize plugins.

Using Qualcomm IM SDK, you can do the following:

- Update or customize the existing plugins for improvements, or make them compatible with the latest version of the GStreamer framework.
- Develop preprocessing and postprocessing plugins for AI/ML models.

6.3.1 Customize a plugin

Customize a Qualcomm GStreamer plugin and install it on the target device.

As an example, the following procedure uses the `mlvdetection` plugin.

1. Set up the environment:

   ```
   cd $ESDK_ROOT
   source environment-setup-armv8-2a-qcom-linux
   ```

   The following message is displayed:

   ```
   SDK environment now set up; additionally you may now run devtool to
   perform development tasks.
   ```

```
Run devtool --help for further details.
[/local/mnt/workspace/QIMSDK_ESDK_6490/]
```

2. Download the Qualcomm IM SDK plugin sources.

```
cd $ESDK_ROOT
```

```
devtool modify qcom-gstreamer1.0-plugins-oss-mlvdetection
```

In this command, `qcom-gstreamer1.0-plugins-oss-mlvdetection` is the BitBake recipe that downloads the sources of the plugin.

```
NOTE: Starting bitbake server...
Loading cache: 100% |#####################################| Time:
0:00:00
Loaded 9816 entries from dependency cache.
Parsing recipes: 100% |#####################################| Time:
0:00:01
Parsing of 6700 .bb  files complete (6678 cached, 22 parsed). 9838
targets, 842 skipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
Initialising tasks: 100% |##################################| Time:
0:00:55
Sstate summary: Wanted 10 local 0 mirrors 0 Missed 10 Current 10 (0%
match, 50% complete)
NOTE: Executing Tasks
NOTE: Tasks Summary: Attempted 93 tasks of which 90 didn't need to be
rerun and all succeeded.
INFO: Source tree extracted to /local/mnt/workspace/QIMSDK_ESDK_6490/
workspace/sources/qcom-gstreamer1.0-plugins-oss-mlvdetection
INFO: Recipe qcom-gstreamer1.0-plugins-oss-mlvdetection now setup to
build from /local/mnt/workspace/QIMSDK_ESDK_6490/workspace/sources/qcom-
gstreamer1.0-plugins-oss-mlvdetection
```

The plugin source code is downloaded to `$ESDK_ROOT/workspace/sources/qcom-gstreamer1.0-plugins-oss-mlvdetection`.

```
[/local/mnt/workspace/QIMSDK_ESDK_6490/workspace/sources/qcom-
gstreamer1.0-plugins-oss-mlvdetection]$ ls
CMakeLists.txt    config.h.in    mlvdetection.h    modules
[/local/mnt/workspace/QIMSDK_ESDK_6490/workspace/sources/qcom-
gstreamer1.0-plugins-oss-mlvdetection]$
```

3. Perform the required customization and run the following command to rebuild the plugin:

```
devtool build qcom-gstreamer1.0-plugins-oss-mlvdetection
```

The following message is displayed:

```
NOTE: Starting bitbake server...
Loading cache: 100% |#################################################|
Time: 0:00:00
Loaded 9816 entries from dependency cache.
Parsing recipes: 100% |#################################################|
Time: 0:00:01
Parsing of 6700 .bb  files complete (6678 cached, 22 parsed). 9838
targets, 842 skipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
Initialising tasks: 100% |##############################################|
Time: 0:00:55
Checking sstate mirror object availability: 100% |####################|
Time: 0:01:06
```

```
Sstate Summary: Wanted 264 Local 0 Mirrors 0 Missed 264 Current 255 (0%
match, 49% complete)
Removing 1 stale sstate objects for arch qcs6490_rb3gen2_vision_kit:
100% |##################################################| Time: 0:00:12
Removing 1 stale sstate objects for arch armv8-2a: 100% |
##################################################################|
Time: 0:00:16
NOTE: Executing Tasks
NOTE: qcom-gstreamer1.0-plugins-oss-mlvdetection: compiling from
external source tree /local/mnt/workspace/QIMSDK_ESDK_6490/workspace/
sources/qcom-gstreamer1.0-plugins-oss-mlvdetection
NOTE: Task Summary: Attempted 1836 tasks of which 1827 didn't need to
be rerun and all succeeded.
```

4. To install the application on the target device, generate the plugin installer (ipk):

   `devtool package qcom-gstreamer1.0-plugins-oss-mlvdetection`

   The installable package is at `ls -1 $ESDK_ROOT/tmp/deploy/ipk/armv8-2a/`.

```
qcom-gstreamer1.0-plugins-oss-mlvdetection_1.0-r0_armv8-2a.ipk
qcom-gstreamer1.0-plugins-oss-mlvdetection_dbg_1.0-r0_armv8-2a.ipk
qcom-gstreamer1.0-plugins-oss-mlvdetection_dev_1.0-r0_armv8-2a.ipk
qcom-gstreamer1.0-plugins-oss-mlvdetection_src_1.0-r0_armv8-2a.ipk
[/local/mnt/workspace/QIMSDK_ESDK_6490/workspace/sources/qcom-
gstreamer1.0-plugins-oss-mlvdetection]$
```

5. Set up a connection with the target device through SSH. For instructions, see Sign in using SSH.

   The development kit is accessible through its configured IP address.

6. Install the compiled reference application on the target device through the SSH shell:

   `ssh root@[IP Address of the device]`

   NOTE If prompted for a password, enter *oelinux123*.

7. On the target device, reconfigure the file system partition to support read and write operations:

   `mount -o remount,rw /`

   `exit`

8. On the host computer, transfer the application installer to the target device:

   `cd $ESDK_ROOT/tmp/deploy/ipk/armv8-2a/`

   `scp qcom-gstreamer1.0-plugins-oss-mlvdetection_1.0-r0_armv8-2a.ipk root@<IP Address of the device>:/opt`

9. Sign in to the SSH shell:

   `ssh root@[IP Address of the device]`

10. On the target device, install the application:

    `opkg --force-depends --force-reinstall --force-overwrite install /opt/qcom-gstreamer1.0-plugins-oss-mlvdetection_1.0-r0_armv8-2a.ipk`

11. Verify if the IPK installation is successful:

```
Installing qcom-gstreamer1.0-plugins-oss-mlvdetection (1.0) on root
Configuring qcom-gstreamer1.0-plugins-oss-mlvdetection.
```

6.3.2 Create a plugin

Create your own plugin and compile it as part of the Qualcomm IM SDK. Each ML plugin is integrated into the SDK as an independent BitBake recipe.

To update the BitBake files for custom plugin compilation, do the following:

1. Set up the environment:

```
cd $ESDK_ROOT

source environment-setup-armv8-2a-qcom-linux

SDK environment now set up; additionally you may now run devtool to
perform development tasks.
Run devtool --help for further details.
[/local/mnt/workspace/Platform_eSDK_plus_QIM]$
```

2. Create a BitBake file at `$ESDK_ROOT/layers/meta-qcom-qim-product-sdk/recipes-gst/gstreamer/`.

   For example:

```
$ESDK_ROOT/layers/meta-qcom-qim-product-sdk/recipes-gst/gstreamer/qcom-
gstreamer1.0-plugins-oss-sample-mlvclassification.bb
```

   a. Copy the contents of the existing plugin to the new file created in step 2.

   b. Update the `SUMMARY`, `SRC_URI`, and `S` variables:

```
...
inherit cmake pkgconfig

SUMMARY = "My custom plugin for ML image categorization"
SECTION= "multimedia"

LICENSE = "BSD-3-Clause-Clear"
LIC_FILES_CHKSUM - "file://${COMMON_LICENSE_DIR}/$
{LICENSE};md5=7a43440b65lf4a472ca93716d01033a"

# Dependencies.
DEPENDS := "gstreamer1.0"
DEPENDS += "gstreamer1.0-plugins-base"
DEPENDS += "qcom-gstreamer1.0-plugins-oss-base"
DEPENDS += "cairo"

#SRCPROJECT = "git://git.codelinaro.org/clo/le/platform/vendor/qcom-
opensource/gst-plugins-qti-oss.git;protocol=https"
#SRCBRANCH  = "imsdk.lnx.2.0.0.r2-rel"
#SRCREV     = "49256b26148d10257959b2b8d55879f72888cb0f"

#SRC_URI = "${SRCPROJECT};branch=${SRCBRANCH};rev=$
{SRCREV};subpath=gst-plugin-mlvclassification"
#S = "${WORKDIR}/gst-plugin-mlvclassification"
SRC_URI += "file://gst-plugin-sample-mlvclassification"
S = "${WORKDIR}/gst-plugin-sample-mlvclassification"

# Instal directories.
INSTALL_INCDIR := "${includedir}"
...
```

3. Create a directory called `files` at `$ESDK_ROOT/layers/meta-qcom-qim-product-sdk/recipes-gst/gstreamer/files`. Under `files`, create the `gst-plugin-sample-mlvclassification` directory, and then copy the Qualcomm IM SDK plugin source code in this directory.

   NOTE Ensure that the source code is available for the `sample-mlvclassification` plugin.

```
files
|
├── gst-plugin-sample-mlvclassification
   |
├── CMakeLists.txt
├── config.h.in
├── mlvclassification.c
├── mlvclassification.h
└── modules
    ├── CMakeLists.txt
    ├── ml-vclassification-mobilenet.c
    ├── ml-vclassification-ocr.c
    ├── ml-vclassification-qfr.c
    └── README
```

4. To include `qcom-gstreamer1.0-plugins-oss-sample-mlvclassification` in the package group, add it under `RDEPENDS` in the `$ESDK_ROOT/layers/meta-qcom-qim-product-sdk/recipes-gst/packagegroups/packagegroup-qcom-gst.bb` directory.

```
RDEPENDS:${PN}:qcom-custom-bsp = " \
    ${PN}-dependencies \
    ${PN}-basic \
    . . .
    qcom-gstreamer1.0-plugins-oss-sample-mlvclassification \
"
```

5. To include `qcom-gstreamer1.0-plugins-oss-sample-mlvclassification:do_package_write_ipk` in package group bbclass, add it under `do_package_write_ipk` in the `$ESDK_ROOT/layers/meta-qcom-qim-product-sdk/classes/qimsdk-pkg.bbclass` directory.

```
GST_PLUGINS = " \
    gstd:do_package_write_ipk \
    gstreamer1.0:do_package_write_ipk \
    . . .
    qcom-gstreamer1.0-plugins-oss-sample-
mlvclassification:do_package_write_ipk \
"
```

6. Download the Qualcomm IM SDK plugin sources:

```
cd $ESDK_ROOT
```

```
devtool modify qcom-gstreamer1.0-plugins-oss-sample-mlvclassification
```

   In this command, `qcom-gstreamer1.0-plugins-oss-sample-mlvclassification` is the BitBake recipe that downloads the plugin source.

```
NOTE: Starting bitbake server ...
Loading cache: 100% |
#######################################################################
############| Time 0:00:02
```

```
Loaded 9814 entries from dependency cache.
Parsing recipes: 100% |
##################################################################
##########| Time 0:00:03
Parsing of 6702 .bb files complete (6670 cached, 32 parsed). 9840
targets, 842 skipped, 0 masked, 0 errors.

Summary: There were 0 WARNING messages.
NOTE: Resolving any missing task queue dependencies
Initialising tasks: 100% |
##################################################################
#######| Time 0:00:08
Sstate summary: Wanted 10 Local 0 Mirrors 0 Missed 10 Current 10 (0%
match, 50% complete)
NOTE: Executing Tasks
NOTE: Tasks Summary: Attempted 93 tasks of which 90 didn't need to be
rerun and all succeeded.
INFO: Source tree extracted to /local/mnt/workspace/QIMSDK_ESDK_6490/
workspace/sources/qcom-gstreamer1.0-plugins-oss-sample-
mlvclassification
INFO: Recipe qcom-gstreamer1.0-plugins-oss-sample-mlvclassification now
set up to build from /local/mnt/workspace/QIMSDK_ESDK_6490/workspace/
sources/qcom-gstreamer1.0-plugins-oss-sample-mlvclassification
```

The plugin source code is downloaded at `$ESDK_ROOT/workspace/sources/qcom-gstreamer1.0-plugins-oss-sample-mlvclassification`.

7. Perform the required customizations and run the following command to rebuild the plugin:

```
devtool build qcom-gstreamer1.0-plugins-oss-sample-mlvclassification
```

```
NOTE: Starting bitbake server ...
Loading cache: 100% |                                        | ETA -:--:--
Loaded 0 entries from dependency cache.
...
```

8. To install the application on the target device, generate the plugin installer (ipk):

```
devtool package qcom-gstreamer1.0-plugins-oss-sample-mlvclassification
```

The installable package is at `$ESDK_ROOT/tmp/deploy/ipk/armv8-2a/`.

```
NOTE: Starting bitbake server ...
Loading cache: 100% |
##################################################################|
ETA 0:00:14
Loaded 9812 entries from dependency cache.
Parsing recipes: 100% |
##################################################################
##########| Time 0:00:03
Parsing of 6702 .bb files complete (6674 cached, 28 parsed). 9840
targets, 842 skipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
Initialising tasks: 100% |
##################################################################
#######| Time 0:00:10
Checking sstate mirror object availability: 100% |
#####################################################| Time 0:00:00
Sstate summary: Wanted 262 Local 0 Mirrors 0 Missed 262 Current 259 (0%
match, 49% complete)
NOTE: Executing Tasks
NOTE: Tasks Summary: Attempted 1847 tasks of which 1847 didn't need to
```

```
be rerun and all succeeded.

Summary: There were 0 WARNING messages.
INFO: Your packages are in /local/mnt/workspace/QIMSDK_ESDK_6490/tmp/
deploy/ipk
```

9. Set up a connection with the target device through SSH. For instructions, see Sign in using SSH.

   Qualcomm Linux is accessible through its configured IP address.

10. On the Qualcomm EVK, install the compiled reference applications:

    ```
    qcom-gstreamer1.0-plugins-oss-sample-mlvclassification_1.0-r0_armv8-2a.ipk
    ```

11. Sign in to the SSH shell:

    ```
    ssh root@[ip-addr]
    ```

    NOTE If prompted for a password, enter *oelinux123*.

12. On the target device, reconfigure the file system partition to support *read* and *write*:

    ```
    mount -o remount,rw /
    ```

    ```
    exit
    ```

13. On the host computer, transfer the application installer to the target device:

    ```
    cd $ESDK_ROOT/tmp/deploy/ipk/armv8-2a/
    ```

    ```
    scp qcom-gstreamer1.0-plugins-oss-sample-mlvclassification_1.0-
    r0_armv8-2a.ipk root@<ip_address>:/opt/
    ```

14. In the SSH shell, run the command on the target device to install the application:

    ```
    opkg --force-depends --force-reinstall --force-overwrite install /opt/qcom-
    gstreamer1.0-plugins-oss-sample-mlvclassification_1.0-r0_armv8-2a.ipk
    ```

7 Use GStreamer utilities for custom plugins

The GStreamer utilities enable you to build your own plugins or GStreamer applications.

The GStreamer framework base provides most of the classes needed for implementation of the custom plugin solutions. The plugins have CPU-based implementations and don't use Qualcomm-specific hardware.

7.1 GSt APIs

These APIs provide the required custom classes and layers to use hardware acceleration in the Qualcomm IM SDK and enable the tensor-based method/workflow for ML.

The table lists all the supported APIs:

Table 7-1 GSt video and ML APIs

API	Description
Video APIs	
GstImageBufferPool APIs	Special GstBufferPool subclass to configure raw video buffers.
GstGlesVideoConverter APIs	Use the GPU through the OpenGL ES based IB2C library to provide the following: ■ Hardware-accelerated image manipulation ■ Transformation ■ Color conversion operations
Machine learning APIs	
GstMLType	Gets information on the GStreamer ML data types.
GstMLInfo	Represents the information structure describing the machine learning properties.
GstMLTensorMeta	Defines the metadata API for different tensor types.
GstMLFrame	Contains mapped pointers to the tensors and the required information about them.
GstMLBufferPool	Special GstBufferPool subclass to configure ML tensor buffers.

7.1.1 Video APIs

Use the video APIs to configure the video buffers and video converter.

GstImageBufferPool APIs

- Configure it to add GstVideoMeta to the buffers.
- Configure video-specific requirements:
  - Stride alignments or pixel padding
  - Type of memory management (GBM/ION).

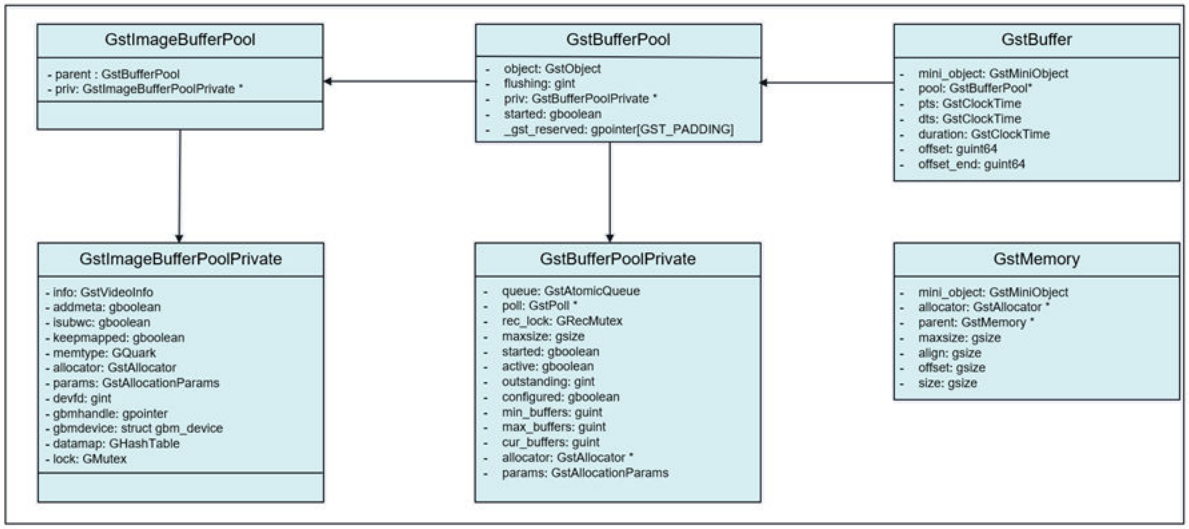


Figure 7-1 GstImageBufferPool workflow

The following list provides the GstImageBufferPool API description:

- **Prototype**

  Creates a buffer pool that can assign the video frame buffers to memory.

  ```
  GstBufferPool *
  gst_image_buffer_pool_new (const gchar * type)
  ```

- **Parameters**

[in]	`type`	The memory that the pool uses to assign buffers.

- **Returns**
  - A new GstBufferPool that allocates the video frames.
  - Free with gst_object_unref.

- **Arguments**

  The following arguments can be passed to`gst_image_buffer_pool_new()`to configure the buffer memories:

Argument	Description	Usage
GST_IMAGE_BUFFER_POOL_TYPE_ION	Assigns buffers through the ION memory manager.	`#define GST_IMAGE_BUFFER_POOL_TYPE_ION "GstBufferPoolTypeIonMemory"`
GST_IMAGE_BUFFER_POOL_TYPE_GBM	Assigns buffers through the generic buffer management (GBM).	`#define GST_IMAGE_BUFFER_POOL_TYPE_GBM "GstBufferPoolTypeGbmMemory"`
GST_IMAGE_BUFFER_POOL_OPTION_UBWC_MODE	Indicates that the allocated buffer must be in Qualcomm® Universal Bandwidth Compression mode.	`#define GST_IMAGE_BUFFER_POOL_OPTION_UBWC_MODE "GstBufferPoolOptionUBWCMode"`
GST_IMAGE_BUFFER_POOL_OPTION_KEEP_MAPPED	Indicates if the mapped buffer memory is mapped until the memory is destroyed.	`#define GST_IMAGE_BUFFER_POOL_OPTION_KEEP_MAPPED "GstBufferPoolOptionKeepMapped"`

GstGlesVideoConverter APIs

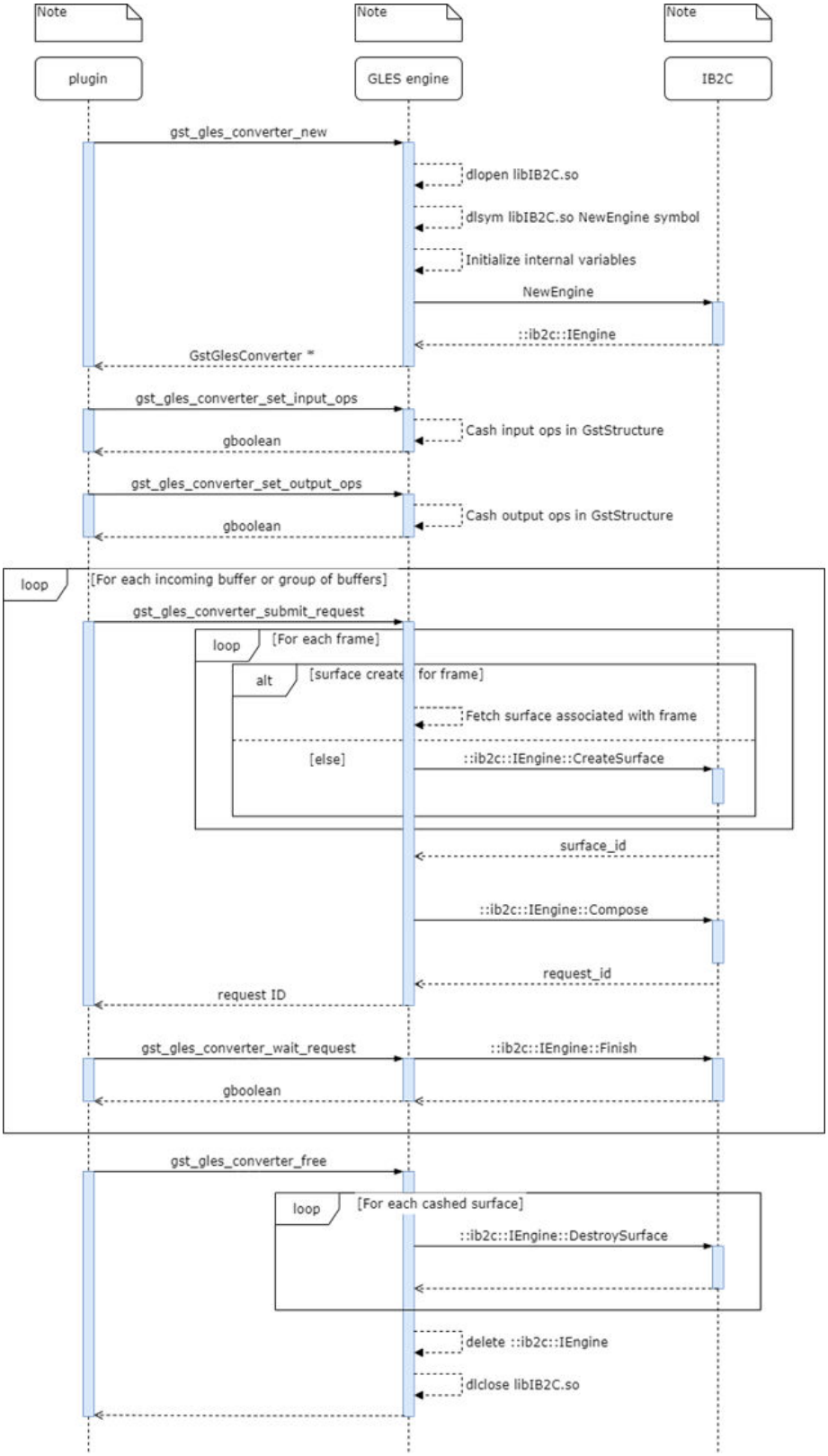


Figure 7-2 Hardware-accelerated image manipulation, transformation, and color conversion

The following list provides information on each of the GstGlesVideoConverter APIs:

- **gst_gles_video_converter_new**

  Creates a GstGlesVideoConverter.

  Prototype

  ```
  GstGlesVideoConverter *
  gst_gles_video_converter_new (void)
  ```

  Returns

  A new GstGlesVideoConverter, free with `gst_gles_video_converter_free()`.

- gst_gles_video_converter_free: Releases all resources held by the converter and frees the memory associated with it.

  Prototype

  ```
  void
  gst_gles_video_converter_free (GstGlesVideoConverter * convert)
  ```

  Parameters

[in]	`convert`	A GstGlesVideoConverter

  Returns

  None

- **gst_gles_video_converter_submit_request**

  Submits a number of video compositions that are executed together.

  Prototype

  ```
  gpointer
          gst_gles_video_converter_submit_request (GstGlesVideoConverter
  *convert, GstGlesComposition * compositions, guint n_compositions)
  ```

  Parameters

[in]	`convert`	A GstGlesVideoConverter
[in]	`compositions`	An array of composition frames
[in]	`n_compositions`	Number of compositions

  Returns

  A unique pointer request ID if the operation was successful.

- **gst_gles_video_converter_wait_request**

  Waits for the requests that have been submitted to GPU compositions to finish.

  Prototype

  ```
  gboolean
  gst_gles_video_converter_wait_request (GstGlesVideoConverter *convert,
  gpointer request_id)
  ```

Parameters

[in]	`convert`	A GstGlesVideoConverter
[in]	`request_id`	Specifies the request ID

Returns

TRUE if the operation was successful.

- **gst_gles_video_converter_flush**

  Waits for the compositions that have been submitted to GPU to finish and flush the data.

  Prototype

```
void
gst_gles_video_converter_flush (GstGlesVideoConverter *convert)
```

  Parameters

[in]	`convert`	A GstGlesVideoConverter

  Returns

  None

7.1.2 Machine learning APIs

Configure the tensor-specific requirements and machine learning properties.

GstMLType

The following are the possible values describing the tensor format:

Enumeration	Description
GST_ML_TYPE_UNKNOWN	Invalid data
GST_ML_TYPE_INT8	Data is represented as 1 byte of signed integer value
GST_ML_TYPE_UINT8	Data is represented as 1 byte of unsigned integer value
GST_ML_TYPE_INT32	Data is represented as 4 bytes of signed integer value
GST_ML_TYPE_UINT32	Data is represented as 4 bytes of unsigned integer value
GST_ML_TYPE_FLOAT16	Data is represented as 2 bytes of floating-point value
GST_ML_TYPE_FLOAT32	Data is represented as 4 bytes of floating-point value

The following list provides information on each of the GstMLType APIs:

- **gst_ml_type_get_size**

  Returns the size of the GstMLType in bytes.

  Prototype

```
guint
gst_ml_type_get_size (GstMLType type)
```

Parameters

[in]	`type`	A GstMLType

Returns

The size in bytes.

- **gst_ml_type_from_string**

  Returns GstMLType based on its string version.

  Prototype

  ```
  GstMLType
  gst_ml_type_from_string (const gchar * type)
  ```

  Parameters

[in]	`type`	A char string version of a GstMLType

  Returns

  A GstMLType

- **gst_ml_type_to_string**

  Returns a char string version of a GstMLType.

  ```
  const gchar *
  gst_ml_type_to_string (GstMLType type)
  ```

  Parameters

[in]	`type`	A string version of a GstMLType

  Returns

  A new char string version of a GstMLType

GstMLInfo

`gst_ml_info_from_caps()` is used to add the information on the ML properties from GstCaps.

Table 7-2 Information structure describing ML properties

Field	Description
`type (GstMLType)`	Type of the tensors
`n_tensors (guint)`	Number of tensors
`n_dimensions (guint)`	Number of dimensions for each tensor
`tensors (guint)`	Array with tensor dimensions

The following list provides information on each of the GstMLInfo APIs:

- **gst_ml_info_init**

  Initializes the GstMLInfo fields.

Prototype

```
void
        gst_ml_info_init (GstMLInfo * info)
```

Parameters

[in]	info	A GstMLInfo

Returns

None

- **gst_ml_info_new**

  Copies a GstMLInfo structure.

  Prototype

  ```
  GstMLInfo *
          gst_ml_info_new (void)
  ```

  Returns

  A new GstMLInfo. Free with `gst_ml_info_free()`.

- gst_ml_info_copy: Copies a GstMLInfo structure.

  Prototype

  ```
  GstMLInfo *
  gst_ml_info_copy (const GstMLInfo * info)
  ```

[in]	info	A GstMLInfo

  Returns

  A new GstMLInfo. Free with `gst_ml_info_free()`.

- **gst_ml_info_free**

  Frees a GstMLInfo structure previously allocated with `gst_ml_info_new()` or `gst_ml_info_copy()`.

  Prototype

  ```
  void
  gst_ml_info_free (const GstMLInfo * info)
  ```

  Parameters

[in]	info	A GstMLInfo

  Returns

  None

- **gst_ml_info_from_caps**

  Parses the capabilities (caps) and updates the `caps` information.

Prototype

```
gboolean
gst_ml_info_from_caps (GstMLInfo * info, const GstCaps  * caps)
```

Parameters

[in]	info	A GstMLInfo
[in]	caps	A GstCaps

Returns

TRUE if the operation was successful

- **gst_ml_info_to_caps**

  Converts the values of info into a GstCaps.

  Prototype

```
GstCaps *
gst_ml_info_to_caps (const GstMLInfo * info)
```

  Parameters

[in]	info	A GstMLInfo

  Returns

  A new GstCaps containing the information.

- **gst_ml_info_is_equal**

  Compares two GstMLInfo and returns whether they're equal.

  Prototype

```
gboolean
gst_ml_info_is_equal (const GstMLInfo * l_info, const GstMLInfo * r_info)
```

  Parameters

[in]	l_info	A GstMLInfo
[in]	r_info	A GstMLInfo

  Returns

  TRUE if the operation was successful.

- **gst_ml_info_tensor_size**

  Calculates the size of the tensor specified by its index from GstMLInfo.

  Prototype

```
gsize
gst_ml_info_tensor_size (const GstMLInfo * info, guint index)
```

Parameters

[in]	`info`	A GstMLInfo
[in]	`index`	The tensor index

Returns

The size in bytes.

- **gst_ml_info_size**

  Calculates the total size of all the tensors inside GstMLInfo.

  Prototype

```
gsize
gst_ml_info_size (const GstMLInfo * info)
```

  Parameters

[in]	`info`	A GstMLInfo

  Returns

  The size in bytes.

GstMLTensorMeta

These APIs are based on GstMeta, which is a member of the GstBuffer metadata structure.

Table 7-3 GstMLTensorMeta information structure

Field	Description
meta (GstMeta)	GstMeta is the parent
id (guint)	ID corresponding to the memory index in GstBuffer
type (GstMLType)	Tensor type
n_dimensions (guint)	Number of tensor dimensions
dimensions (guint)	Array of tensor dimensions

The following list provides information on each of the GstMLTensorMeta APIs:

- **gst_buffer_add_ml_tensor_meta**

  Attaches GstMLTensorMeta metadata to a buffer with the given parameters.

  Prototype

```
GstMLTensorMeta *
gst_buffer_add_ml_tensor_meta (GstBuffer * buffer, const GstMLType type,
const guint n_dimensions, const guint dimensions[GST_ML_TENSOR_MAX_DIMS])
```

  Parameters

[in]	`buffer`	A GstBuffer
[in]	`type`	The tensor type.
[in]	`n_dimensions`	The number of tensor dimensions.
[in]	`dimensions`	The array that has the tensor dimensions.

Returns

The GstMLTensorMeta on buffer. Don't free after the code is run.

- **gst_buffer_get_ml_tensor_meta**

  Gets the first GstMLTensorMeta on buffer.

  Prototype

  ```
  GstMLTensorMeta *
  gst_buffer_get_ml_tensor_meta (GstBuffer * buffer)
  ```

  Parameters

[in]	buffer	A GstBuffer

  Returns

  The GstMLTensorMeta on buffer. Don't free after the code is run.

- **gst_buffer_get_ml_tensor_meta_id**

  Finds the GstMLTensorMeta on buffer with the given ID.

  Prototype

  ```
  GstMLTensorMeta *
  gst_buffer_get_ml_tensor_meta_id (GstBuffer * buffer, guint id)
  ```

  Parameters

[in]	buffer	A GstBuffer
[in]	id	A metadata ID

  Returns

  The GstMLTensorMeta on buffer. Don't free after the code is run.

- **gst_ml_meta_tensor_size**

  Gets the total size of the tensor in bytes, which is calculated based on the dimensions and tensor type.

  Prototype

  ```
  gsize
  gst_ml_meta_tensor_size (const GstMLTensorMeta * meta)
  ```

  Parameters

[in]	meta	A GstMLTensorMeta

  Returns

  The size in bytes.

GstMLFrame

GstMLFrame is a structure obtained from `gst_ml_frame_map()`.

Table 7-4 Structure of GstMLFrame

Field	Description
info (GstMLInfo)	The GstMLInfo
buffer (GstBuffer)	Mapped buffer containing the tensor memory blocks
map (GstMapInfo)	Mappings of the tensor memory blocks

The following list provides information on each of the GstMLFrame APIs:

- **gst_ml_frame_map**

  Updates the frame values using GstMLInfo and GstBuffer.

  a. The address is passed to the GstMLFrame structure that's allocated on the stack.

  b. The `gst_ml_frame_map()` function fills the GstMLFrame structure with ML tensor-specific information to access the ML buffer data. All the buffer video tensors are mapped and the pointers are set in the frame data.

  Prototype

  ```
  gboolean
  gst_ml_frame_map (GstMLFrame * frame, const GstMLInfo * info, GstBuffer *
  buffer, GstMapFlags flags)
  ```

  Parameters

[in]	`frame`	Pointer to GstMLFrame
[in]	`info`	A GstMLInfo
[in]	`buffer`	A GstBuffer
[in]	`flags`	A GstMapFlags

  Returns

  TRUE if the map operation was successful.

- **gst_ml_frame_unmap**:

  Un-maps the memory that was earlier mapped with `gst_ml_frame_map()`.

  Prototype

  ```
  void
  gst_ml_frame_unmap (GstMLFrame * frame)
  ```

  Parameters

[in]	`frame`	Pointer to GstMLFrame

  Returns

  None

GstMLBufferPool

Configure the following using GstMLBufferPool API:

- Tensor-specific requirements such as several GstMemory blocks in a single GstBuffer

  Type of memory used for allocation.
- Addition of GstMLTensorMeta to the buffers.

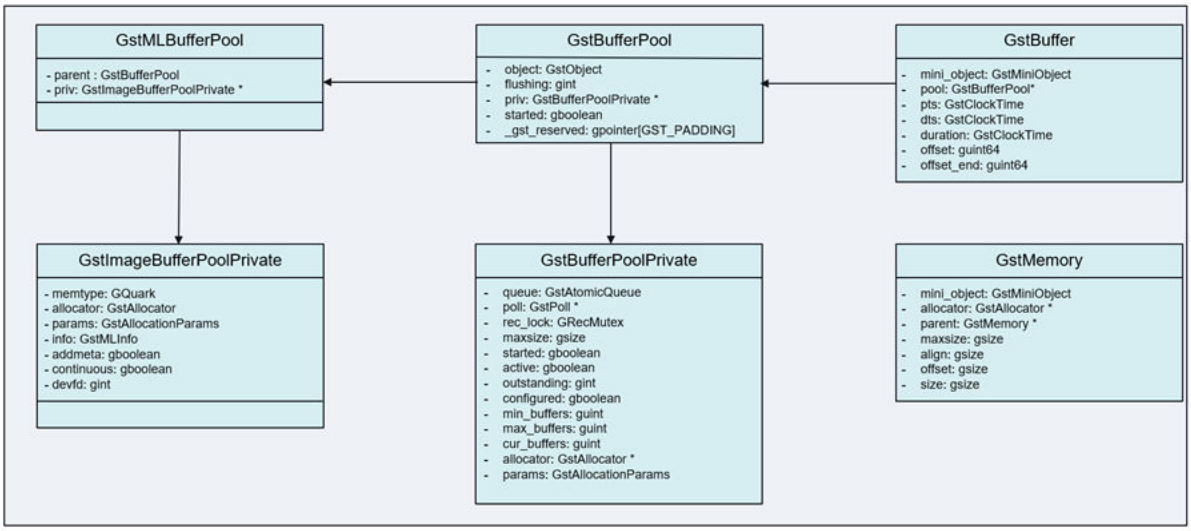


Figure 7-3 GstBufferPool subclass for ML

- **Prototype**

  Creates a buffer pool that can assign the ML frames.

  ```
  GstBufferPool *
  gst_ml_buffer_pool_new (const gchar * type)
  ```

- **Parameters**

[in]	`type`	The memory used by the pool to assign buffers.

- **Returns**
  - A new GstBufferPool to assign ML frames.
  - Free with gst_object_unref.
- **Arguments**

  The following arguments can be passed to `gst_ml_buffer_pool_new()` to configure the buffer memories:

Argument	Description	Usage
GST_ML_BUFFER_POOL_TYPE_ION	Assigns buffers by passing it as arguments to `gst_ml_buffer_pool_new()`	`#define GST_ML_BUFFER_POOL_TYPE_ION "GstMLBufferPoolTypeIonMemory"`
GST_ML_BUFFER_POOL_TYPE_SYSTEM		`#define GST_ML_BUFFER_POOL_TYPE_SYSTEM "GstMLBufferPoolTypeSystemMemory"`

Argument	Description	Usage
GST_ML_BUFFER_POOL_OPTION_TENSOR_META	Requests ML tensor metadata on buffers from the pool.	`#define GST_ML_BUFFER_POOL_OPTION_TENSOR_META "GstMLBufferPoolOptionTensorMeta"`
GST_ML_BUFFER_POOL_OPTION_CONTINUOUS	Requests all tensors to be into a continuous physical memory.	`#define GST_ML_BUFFER_POOL_OPTION_CONTINUOUS "GstMLBufferPoolOptionContinuous"`

7.2 GSt debug

GST_DEBUG is a debugging technique where a specified number sets the appropriate level for logging.

The `GST_DEBUG` environment variable controls the GStreamer debug output.

For example, to log all warnings, run the following command:

```
export GST_DEBUG=2
```

7.3 GStD framework

GStreamer daemon (GStD) controls the audio-video streaming using an inter-process communication protocol (IPC).

- It's an enhancement to the gst-launch tool.
- It allows you to create a pipeline, and control the settings such as play and pause when the pipeline is active.

GStD deploys in two parts:

- **Daemon** starts the process in the background.
- **Control application**, which can be written in any language, can run on the same processor or a separate machine.

Known issue

GStD is currently not supported on QCS9075 and QCS8275.

GStD project

The GStD project includes the core, the GstdClient, and IPC interfaces.

GStD provides core and IPC endpoint.

GStD design components	Description
Core	Holds the current state of the GStreamer daemon pipelines along with their respective states, elements in each pipeline, and properties of each element.
IPC	▪ Allows client applications to change the GStD core state using an IPC mechanism such as TCP, HTTP, DBus, sockets. ▪ Allows the custom applications to change the state of the GStD core. For example, IPC is used to do the following: ▫ Create a pipeline. ▫ Set the state to play. ▫ Change an element property while the pipeline is running.

The custom application provides view and client logic.

Custom application design	Description
View	▪ Provides visual feedback of the GStD core state. ▪ Represents a graphical user interface, a web page, a command-line application. ▪ Allows you to see the view that's updated by the model.
Client logic	▪ Provides the custom functionality of the application. ▪ Allows you to use IPC to change the GStD core or model, and receive feedback through the GUI/view.

A GStreamer client (GstdClient) is distributed with the project. It's a command-line based application that communicates with GStD through TCP.

- GstdClient allows you to create and start pipelines.
- Unlike `gst-launch`, GstdClient allows you to have several active pipelines with the ability to control the pipeline and receive feedback after the pipeline has been created.

For more information about GStD, see https://developer.ridgerun.com/wiki/index.php/GStreamer_Daemon.

Run GStD

1. Start GStD:

```
gstd
# This will start the process in the background. Using 'gstd &' is not
required becauses gstd daemonizes itself by default.

# To run gstd in foreground
gstd -D

# To view a list of all supported options,
gstd --help-all
```

2. Start GstdClient:

```
gst-client
# gstd prompt will appear like,
# gstd>
# In the prompt, we can interact with the server creating/playing
pipelines, etc.

# To list all the possible commands
gstd> help

# Alternatively, instead of opening gstd prompt, can give the command as
argument to gst-client
gst-client list_pipelines
```

7.3.1 Run local use cases

Control the GStD framework using a control application that's running on same processor.

Prerequisites

Enable display:

```
vi /etc/default/gstd
     OPTARGS=" -a 0.0.0.0"

     XDG_RUNTIME_DIR=/dev/socket/weston

     WAYLAND_DISPLAY=wayland-1

systemctl daemon-reload
systemctl restart gstd
```

Single stream encode

1. Create the pipeline:

```
gst-client-1.0 pipeline_create enc1 qtiqmmfsrc name=camsrc ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4 output-
io-mode=5 ! h264parse ! queue ! mp4mux ! queue ! filesink location=/tmp/
enc1.mp4
```

2. Control the pipeline:

```
gst-client-1.0 pipeline_play enc1
```

   - Send end of stream (EOS):

```
gst-client-1.0 event_eos enc1
```

   - Stop the pipeline:

```
gst-client-1.0 pipeline_stop enc1
```

Multistream encode

1. Create the pipeline:

```
gst-client-1.0 pipeline_create enc2 qtiqmmfsrc name=camsrc
video_0::type=preview ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! queue ! v4l2h264enc capture-io-mode=4
output-io-mode=5 ! queue ! h264parse ! mp4mux ! queue ! filesink
location="/tmp/mux1.mp4" \ camsrc. ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! queue ! v4l2h264enc capture-io-mode=4
output-io-mode=5 ! queue ! h264parse ! mp4mux ! queue ! filesink
location="/tmp/mux2.mp4"
```

2. Control the pipeline:

```
gst-client-1.0 pipeline_play enc2
```

   - Send EOS:

```
gst-client-1.0 event_eos enc2
```

   - Stop the pipeline:

```
gst-client-1.0 pipeline_stop enc2
```

Single display

1. Create the pipeline:

```
gst-client-1.0 pipeline_create dis1 qtiqmmfsrc name=camsrc
video_0::type=preview ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1 ! waylandsink
fullscreen=true async=true sync=false
```

2. Control the pipeline:

   - Play the pipeline:

```
gst-client-1.0 pipeline_play dis1
```

   - Pause the pipeline:

```
gst-client-1.0 pipeline_pause dis1
```

   - Stop the pipeline:

```
gst-client-1.0 pipeline_stop dis1
```

Multi-display

1. Create the pipeline:

```
gst-client-1.0 pipeline_create dis2 qtiqmmfsrc name=camsrc
video_0::type=preview ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1 ! waylandsink x=0
y=0 width=500 height=400 async=true sync=false camsrc. ! video/x-
```

```
raw,format=NV12,width=1920,height=1080,framerate=30/1 ! waylandsink x=510
y=0 width=500 height=400 async=true sync=false
```

2. Control the pipeline:
   - Play the pipeline:
     ```
     gst-client-1.0 pipeline_play dis2
     ```
   - Pause the pipeline:
     ```
     gst-client-1.0 pipeline_pause dis2
     ```
   - Stop the pipeline:
     ```
     gst-client-1.0 pipeline_stop dis2
     ```

7.3.2 Run TCP use cases

Control the GStD framework using a control application that's remotely interfacing through TCP.

Prerequisites

- Enable `wlan0`. For instructions, see Sign in using SSH.
- After the system restart, the Wi-Fi should be up and running.
  ```
  ifconfig   // Note down wlan0_IP
  ```
- To configure TCP use cases, select an unused port:
  ```
  vi /etc/default/gstd
       OPTARGS=" -a wlan0_IP -p 5001"     // Replace the port 5001 if the
  port is not free

       XDG_RUNTIME_DIR=/dev/socket/weston

       WAYLAND_DISPLAY=wayland-1

  systemctl daemon-reload
  systemctl restart gstd
  ```

Single stream encode

1. Create the pipeline:
   ```
   gst-client-1.0 -a 10.92.164.1 -p 5001 pipeline_create tcp_enc1 qtiqmmfsrc
   name=camsrc ! video/x-
   raw,format=NV12,width=1920,height=1080,framerate=30/1,interlace-
   mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4 output-
   io-mode=5 ! h264parse ! queue ! mp4mux ! queue ! filesink location=/tmp/
   tcp_enc1.mp4
   ```
2. Control the pipeline:
   - Play the pipeline:
     ```
     gst-client-1.0 -a 10.92.164.1 -p 5001 pipeline_play tcp_enc1
     ```

- Send EOS:

  ```
  gst-client-1.0 -a 10.92.164.1 -p 5001 event_eos tcp_enc1
  ```

- Stop the pipeline:

  ```
  gst-client-1.0 -a 10.92.164.1 -p 5001 pipeline_stop tcp_enc1
  ```

Multistream encode

1. Create the pipeline:

   ```
   gst-client-1.0 -a 10.92.164.1 -p 5001 pipeline_create tcp_enc2 qtiqmmfsrc
   name=camsrc video_0::type=preview ! video/x-
   raw,format=NV12,width=1920,height=1080,framerate=30/1,interlace-
   mode=progressive,colorimetry=bt601 ! queue ! v4l2h264enc capture-io-mode=4
   output-io-mode=5 ! queue ! h264parse ! mp4mux ! queue ! filesink
   location="/tmp/tcp_mux1.mp4" \ camsrc. ! video/x-
   raw,format=NV12,width=1920,height=1080,framerate=30/1,interlace-
   mode=progressive,colorimetry=bt601 ! queue ! v4l2h264enc capture-io-mode=4
   output-io-mode=5 ! queue ! h264parse ! mp4mux ! queue ! filesink
   location="/tmp/tcp_mux2.mp4"
   ```

2. Control the pipeline:

   - Play the pipeline:

     ```
     gst-client-1.0 -a 10.92.164.1 -p 5001 pipeline_play tcp_enc2
     ```

   - Send EOS:

     ```
     gst-client-1.0 -a 10.92.164.1 -p 5001 event_eos tcp_enc2
     ```

   - Stop the pipeline:

     ```
     gst-client-1.0 -a 10.92.164.1 -p 5001 pipeline_stop tcp_enc2
     ```

Single display

1. Create the pipeline:

   ```
   gst-client-1.0 -a 10.92.164.1 -p 5001 pipeline_create tcp_dis1 qtiqmmfsrc
   name=camsrc video_0::type=preview ! video/x-
   raw,format=NV12,width=1920,height=1080,framerate=30/1 ! waylandsink
   fullscreen=true async=true sync=false
   ```

2. Control the pipeline:

   - Play the pipeline:

     ```
     gst-client-1.0 -a 10.92.164.1 -p 5001 pipeline_play tcp_dis1
     ```

   - Pause the pipeline:

     ```
     gst-client-1.0 -a 10.92.164.1 -p 5001 pipeline_pause tcp_dis1
     ```

   - Stop the pipeline:

     ```
     gst-client-1.0 -a 10.92.164.1 -p 5001 pipeline_stop tcp_dis1
     ```

Multi-display

1. Create the pipeline:

```
gst-client-1.0 -a 10.92.164.1 -p 5001 pipeline_create tcp_dis2 qtiqmmfsrc
name=camsrc video_0::type=preview ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1 ! waylandsink x=0
y=0 width=500 height=400 async=true sync=false camsrc. ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1 ! waylandsink x=510
y=0 width=500 height=400 async=true sync=false
```

2. Control the pipeline:
   - Play the pipeline:

     ```
     gst-client-1.0 -a 10.92.164.1 -p 5001 pipeline_play tcp_dis2
     ```

   - Pause the pipeline:

     ```
     gst-client-1.0 -a 10.92.164.1 -p 5001 pipeline_pause tcp_dis2
     ```

   - Stop the pipeline:

     ```
     gst-client-1.0 -a 10.92.164.1 -p 5001 pipeline_stop tcp_dis2
     ```

7.3.3 Run HTTP use cases

Control the GStD framework using a control application that's remotely interfacing through HTTP.

Prerequisites

- Enable `wlan0`. For instructions, see Sign in using SSH.
- After the system restart, the Wi-Fi should be up and running.

  ```
  ifconfig   // Note down wlan0_IP
  ```

- To configure HTTP use cases, select an unused port:

  ```
  vi /etc/default/gstd
       OPTARGS=" --enable-http-protocol --http-address=<wlan0_IP_address> --
  http-port=5002"       // Replace the port 5002 if the port is not free;
  For <wlan0_IP_address>, enter your device WLAN IP address

       XDG_RUNTIME_DIR=/dev/socket/weston

       WAYLAND_DISPLAY=wayland-1

  systemctl daemon-reload
  systemctl restart gstd
  ```

- Download Postman to run the HTTP use cases.

Single stream encode

1. Create the pipeline–POST:

```
http://<wlan0_IP_address>:5002/pipelines?
name=pipeline_http_enc1&description=qtiqmmfsrc name=camsrc !
video,format=NV12,width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! v4l2h264enc capture-io-mode=4 output-
io-mode=5 ! h264parse ! queue ! mp4mux ! queue ! filesink location=/tmp/
http_enc1.mp4
```

2. Control the pipeline:
   - Play the pipeline–PUT:

     ```
     http://<wlan0_IP_address>:5002/pipelines/pipeline_http_enc1/state?
     name=playing
     ```

   - Send EOS–POST:

     ```
     http://<wlan0_IP_address>:5002/pipelines/pipeline_http_enc1/event?
     name=eos
     ```

   - Stop the pipeline–PUT:

     ```
     http://<wlan0_IP_address>:5002/pipelines/pipeline_http_enc1/state?
     name=null
     ```

Multistream encode

1. Create the pipeline–POST

```
http://<wlan0_IP_address>:5002/pipelines?
name=pipeline_http_enc2&description=qtiqmmfsrc name=camsrc
video_0::type=preview ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! queue ! v4l2h264enc capture-io-mode=4
output-io-mode=5 ! queue ! h264parse ! mp4mux ! queue ! filesink
location=/tmp/http_mux1.mp4 camsrc. ! video/x-
raw,format=NV12,width=1920,height=1080,framerate=30/1,interlace-
mode=progressive,colorimetry=bt601 ! queue ! v4l2h264enc capture-io-mode=4
output-io-mode=5 ! queue ! h264parse ! mp4mux ! queue ! filesink
location=/tmp/http_mux2.mp4
```

2. Control pipeline:
   - Play the pipeline–PUT:

     ```
     http://<wlan0_IP_address>:5002/pipelines/pipeline_http_enc2/state?
     name=playing
     ```

   - Send EOS–POST:

     ```
     http://<wlan0_IP_address>:5002/pipelines/pipeline_http_enc2/event?
     name=eos
     ```

- Stop the pipeline–PUT:

  ```
  http://<wlan0_IP_address>:5002/pipelines/pipeline_http_enc2/state?
  name=null
  ```

Single display

1. Create the pipeline–POST:

   ```
   http://<wlan0_IP_address>:5002/pipelines?
   name=pipeline_http_dis1&description=qtiqmmfsrc name=camsrc
   video_0::type=preview ! video/x-
   raw,format=NV12,width=1920,height=1080,framerate=30/1 ! waylandsink
   fullscreen=true async=true sync=false
   ```

2. Control the pipeline:
   - Play the pipeline–PUT:

     ```
     http://<wlan0_IP_address>:5002/pipelines/pipeline_http_dis1/state?
     name=playing
     ```

   - Send EOS–POST:

     ```
     http://<wlan0_IP_address>:5002/pipelines/pipeline_http_dis1/event?
     name=eos
     ```

   - Stop the pipeline–PUT:

     ```
     http://<wlan0_IP_address>:5002/pipelines/pipeline_http_dis1/state?
     name=null
     ```

Multi-display

1. Create the pipeline–POST:

   ```
   http://<wlan0_IP_address>:5002/pipelines?
   name=pipeline_http_dis2&description=qtiqmmfsrc name=camsrc
   video_0::type=preview ! video/x-
   raw,format=NV12,width=1920,height=1080,framerate=30/1 ! waylandsink x=0
   y=0 width=500 height=400 async=true sync=false camsrc. ! video/x-
   raw,format=NV12,width=1920,height=1080,framerate=30/1 ! waylandsink x=510
   y=0 width=500 height=400 async=true sync=false
   ```

2. Control the pipeline:
   - Play the pipeline–PUT:

     ```
     http://<wlan0_IP_address>:5002/pipelines/pipeline_http_dis2/state?
     name=playing
     ```

   - Send EOS–POST:

     ```
     http://<wlan0_IP_address>:5002/pipelines/pipeline_http_dis2/event?
     name=eos
     ```

- Stop the pipeline–PUT:

  ```
  http://<wlan0_IP_address>:5002/pipelines/pipeline_http_dis2/state?
  name=null
  ```

7.3.4 Enable message support through MQTT

Message Queuing Telemetry Transport (MQTT) is a lightweight publish-subscribe protocol for communication between devices. Qualcomm IM SDK supports the MQTT protocol to send messages to a host device.

After establishing communication with an MQTT server, the `qtimsghub` component of Qualcomm IM SDK can be used to publish messages or subscribe to messages from the server machine.

Prerequisites

Do the following to enable an M2M session between an MQTT server running on the Linux host computer and the Qualcomm EVK acting as the client.

NOTE These use cases require a functional MQTT server.

1. Install and configure the Mosquitto server on a Linux machine.
   a. Install the Mosquitto server.

      ```
      sudo apt-get install mosquitto
      ```

      For more information, see Download | Eclipse Mosquitto.

      NOTE If you don't need to set a username and password, you can skip steps (b) and (c), and proceed with step (d).

   b. Create the username and password on the server side.

      ```
      sudo mosquitto_passwd -c /etc/mosquitto/pwfile <username>
      ```

   c. Configure the Mosquitto server.
      i. Add the following text in the `/etc/mosquitto/mosquitto.conf` file.

         ```
         password_file /etc/mosquitto/pwfile
         allow_anonymous false
         listener 1883
         ```

   d. Start the Mosquitto server.

      ```
      mosquitto -c <config file path to start mosquitto server> (another
      method: systemctl restart mosquitto)
      ```

2. Create a plugin (client) configuration file on the client side (Qualcomm EVK).

   Create the configuration file at `path /usr/local/`, and add the following text in the configuration file.

```
 mqtt_version = MQTTV311
username = <username set using mosquitto_passwd before>
 password = <password set using mosquitto_passwd before>
```

```
sh-5.1# cat /usr/local/config
# clean_session = TRUE
# keepalive = 60
# retain = FALSE

# mqtt_version: MQTTV31, MQTTV311, MQTTV5
# mqtt_version = MQTTV311

# will_topic = NULL
# will_payload = NULL
# will_qos = 0
# will_retain = FALSE
# max_inflight = 20
# tcp_nodelay = FALSE
# username = NULL
# password = NULL
# usd_path = NULL
# socks5_host = NULL
# socks5_port = 1883
# socks5_password = NULL
# mosquitto_property = NULL

# mqtt_version: MQTTV31, MQTTV311, MQTTV5
mqtt_version = MQTTV311

username = mohanga
password = 12345
```

3. Ensure that the device and Linux machine (server) are in the same network, and they can communicate with each other.

Message in command line with topic

Before you run the commands, ensure that the configuration file is at the appropriate path.

1. Subscriber:

```
gst-launch-1.0 -e qtimsgsub topic=tmp host=<mosquitto server ip> port=1883
protocol=mqtt config=/usr/local/config ! filesink location=/opt/
sub_message.txt
```

2. Publisher:

```
gst-launch-1.0 -e fakesrc ! qtimsgpub topic=tmp message="Tmp Message"
host=<mosquitto server ip> port=1883 protocol=mqtt config=/usr/local/config
```

To stop the use case, use **CTRL + C**.

Message in file with topic

Before you run the commands, ensure the following:

- The configuration file is at the appropriate path.
- The `message.txt` file has content that can be published.

1. Subscriber:

```
gst-launch-1.0 -e qtimsgsub topic=tmp host=<mosquitto server ip> port=1883
protocol=mqtt config=/usr/local/config ! filesink location=/opt/
sub_message.txt
```

2. Publisher:

```
echo "Message in file" > /usr/local/message.txt
```

```
gst-launch-1.0 -e filesrc location=/usr/local/message.txt ! qtimsgpub
topic=tmp host=<mosquitto server ip> port=1883 protocol=mqtt config=/usr/
local/config
```

To stop the use case, use **CTRL + C**.

Message in pipeline–JSON format

Before you run the commands, ensure the following:

- The configuration file is at the appropriate path.
- The AI model, engine, and an MP4 file are available.
- Configure display:

```
export XDG_RUNTIME_DIR=/dev/socket/weston && export
WAYLAND_DISPLAY=wayland-1
```

- Push the models and label files to your device. For more information, see Download model and label files.

1. Subscriber:

```
gst-launch-1.0 -e qtimsgsub topic=tmp host=<mosquitto server ip> port=1883
protocol=mqtt config=/usr/local/config ! filesink location=/usr/local/
sub.json async=false
```

2. Publisher:

```
gst-launch-1.0 -e  filesrc location=/etc/media/<h264_file>.mp4 ! qtdemux !
queue ! h264parse ! v4l2h264dec capture-io-mode=4 output-io-mode=4 !
queue ! qtimlvconverter ! queue ! qtimltflite delegate=external external-
delegate-path=libQnnTFLiteDelegate.so external-delegate-
options="QNNExternalDelegate,backend_type=htp;"  model=/etc/models/local/
mobilenet_v2_1.0_224_quant.tflite ! queue ! qtimlvclassification
threshold=60.0 results=3 module=mobilenet labels=/etc/labels/local/
mobilenet.labels constants="Mobilenet,q-offsets=<58.0>,q-
scales=<0.09889253973960876>;" ! text/x-raw ! queue ! qtimsgpub topic=tmp
```

```
host=<client-ip-address> port=1883 protocol=mqtt config=/etc/configs/local/
config json=true async=false
```

To stop a use case, use **CTRL + C**.

8 Appendix: Qualcomm IM SDK plugin sizes

The table lists the sizes of the plugin release, developer, and debug variants.

Plugin details	Size in MB
Release variant	**69.96**
Upstream plugins	11.75
Qualcomm GST plugins	58.21
Developer variant	**3.22**
Upstream plugins	1.18
Qualcomm GST plugins	2.04
Debug variant	**86.26**
Upstream plugins	65.31
Qualcomm GST plugins	20.94

LEGAL INFORMATION

Your access to and use of this material, along with any documents, software, specifications, reference board files, drawings, diagnostics and other information contained herein (collectively this "Material"), is subject to your (including the corporation or other legal entity you represent, collectively "You" or "Your") acceptance of the terms and conditions ("Terms of Use") set forth below. If You do not agree to these Terms of Use, you may not use this Material and shall immediately destroy any copy thereof.

1) Legal Notice.

This Material is being made available to You solely for Your internal use with those products and service offerings of Qualcomm Technologies, Inc. ("**Qualcomm Technologies**"), its affiliates and/or licensors described in this Material, and shall not be used for any other purposes. If this Material is marked as "**Qualcomm Internal Use Only**", no license is granted to You herein, and You must immediately (a) destroy or return this Material to Qualcomm Technologies, and (b) report Your receipt of this Material to qualcomm.support@qti.qualcomm.com. This Material may not be altered, edited, or modified in any way without Qualcomm Technologies' prior written approval, nor may it be used for any machine learning or artificial intelligence development purpose which results, whether directly or indirectly, in the creation or development of an automated device, program, tool, algorithm, process, methodology, product and/or other output. Unauthorized use or disclosure of this Material or the information contained herein is strictly prohibited, and You agree to indemnify Qualcomm Technologies, its affiliates and licensors for any damages or losses suffered by Qualcomm Technologies, its affiliates and/or licensors for any such unauthorized uses or disclosures of this Material, in whole or part.

Qualcomm Technologies, its affiliates and/or licensors retain all rights and ownership in and to this Material. No license to any trademark, patent, copyright, mask work protection right or any other intellectual property right is either granted or implied by this Material or any information disclosed herein, including, but not limited to, any license to make, use, import or sell any product, service or technology offering embodying any of the information in this Material.

THIS MATERIAL IS BEING PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, WHETHER EXPRESSED, IMPLIED, STATUTORY OR OTHERWISE. TO THE MAXIMUM EXTENT PERMITTED BY LAW, QUALCOMM TECHNOLOGIES, ITS AFFILIATES AND/OR LICENSORS SPECIFICALLY DISCLAIM ALL WARRANTIES OF TITLE, MERCHANTABILITY, NON-INFRINGEMENT, FITNESS FOR A PARTICULAR PURPOSE, SATISFACTORY QUALITY, COMPLETENESS OR ACCURACY, AND ALL WARRANTIES ARISING OUT OF TRADE USAGE OR OUT OF A COURSE OF DEALING OR COURSE OF PERFORMANCE. MOREOVER, NEITHER QUALCOMM TECHNOLOGIES, NOR ANY OF ITS AFFILIATES AND/OR LICENSORS, SHALL BE LIABLE TO YOU OR ANY THIRD PARTY FOR ANY EXPENSES, LOSSES, USE, OR ACTIONS HOWSOEVER INCURRED OR UNDERTAKEN BY YOU IN RELIANCE ON THIS MATERIAL.

Certain product kits, tools and other items referenced in this Material may require You to accept additional terms and conditions before accessing or using those items.

Technical data specified in this Material may be subject to U.S. and other applicable export control laws. Transmission contrary to U.S. and any other applicable law is strictly prohibited.

Nothing in this Material is an offer to sell any of the components or devices referenced herein.

This Material is subject to change without further notification.

In the event of a conflict between these Terms of Use and the *Website Terms of Use* on www.qualcomm.com, the *Qualcomm Privacy Policy* referenced on www.qualcomm.com, or other legal statements or notices found on prior pages of the Material, these Terms of Use will control. In the event of a conflict between these Terms of Use and any other agreement (written or click-through, including, without limitation any non-disclosure agreement) executed by You and Qualcomm Technologies or a Qualcomm Technologies affiliate and/or licensor with respect to Your access to and use of this Material, the other agreement will control.

These Terms of Use shall be governed by and construed and enforced in accordance with the laws of the State of California, excluding the U.N. Convention on International Sale of Goods, without regard to conflict of laws principles. Any dispute, claim or controversy arising out of or relating to these Terms of Use, or the breach or validity hereof, shall be adjudicated only by a court of competent jurisdiction in the county of San Diego, State of California, and You hereby consent to the personal jurisdiction of such courts for that purpose.

2) Trademark and Product Attribution Statements.

Qualcomm is a trademark or registered trademark of Qualcomm Incorporated. Arm is a registered trademark of Arm Limited (or its subsidiaries) in the U.S. and/or elsewhere. The Bluetooth® word mark is a registered trademark owned by Bluetooth SIG, Inc. Other product and brand names referenced in this Material may be trademarks or registered trademarks of their respective owners.

Snapdragon and Qualcomm branded products referenced in this Material are products of Qualcomm Technologies, Inc. and/or its subsidiaries. Qualcomm patented technologies are licensed by Qualcomm Incorporated.