SFML

* [Main Page](http://docs.google.com/index.htm)
* [Modules](http://docs.google.com/modules.htm)
* [Classes](http://docs.google.com/annotated.htm)
* [Files](http://docs.google.com/files.htm)
* [File List](http://docs.google.com/files.htm)
* [include](http://docs.google.com/dir_f3190241575fd2bd132a392ae6942f4a.htm)
* [SFML](http://docs.google.com/dir_692f376662c82a26cfe4cfa3aceebe24.htm)
* [System](http://docs.google.com/dir_60c5c649f8df3b69a45a020d59f81335.htm)

Utf.inl

1

2 //

3 // SFML - Simple and Fast Multimedia Library

4 // Copyright (C) 2007-2012 Laurent Gomila (laurent.gom@gmail.com)

5 //

6 // This software is provided 'as-is', without any express or implied warranty.

7 // In no event will the authors be held liable for any damages arising from the use of this software.

8 //

9 // Permission is granted to anyone to use this software for any purpose,

10 // including commercial applications, and to alter it and redistribute it freely,

11 // subject to the following restrictions:

12 //

13 // 1. The origin of this software must not be misrepresented;

14 // you must not claim that you wrote the original software.

15 // If you use this software in a product, an acknowledgment

16 // in the product documentation would be appreciated but is not required.

17 //

18 // 2. Altered source versions must be plainly marked as such,

19 // and must not be misrepresented as being the original software.

20 //

21 // 3. This notice may not be removed or altered from any source distribution.

22 //

24

25

27 // References :

28 //

29 // http://www.unicode.org/

30 // http://www.unicode.org/Public/PROGRAMS/CVTUTF/ConvertUTF.c

31 // http://www.unicode.org/Public/PROGRAMS/CVTUTF/ConvertUTF.h

32 // http://people.w3.org/rishida/scripts/uniview/conversion

33 //

35

36

38 template <typename In>

39 In [Utf<8>::decode](http://docs.google.com/classUtf.htm)(In begin, In end, Uint32& output, Uint32 replacement)

40 {

41  // Some useful precomputed data

42  static const int trailing[256] =

43  {

44  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

45  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

46  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

47  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

48  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

49  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

50  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

51  2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5

52  };

53  static const Uint32 offsets[6] =

54  {

55  0x00000000, 0x00003080, 0x000E2080, 0x03C82080, 0xFA082080, 0x82082080

56  };

57

58  // decode the character

59  int trailingBytes = trailing[static\_cast<Uint8>(\*begin)];

60  if (begin + trailingBytes < end)

61  {

62  output = 0;

63  switch (trailingBytes)

64  {

65  case 5 : output += static\_cast<Uint8>(\*begin++); output <<= 6;

66  case 4 : output += static\_cast<Uint8>(\*begin++); output <<= 6;

67  case 3 : output += static\_cast<Uint8>(\*begin++); output <<= 6;

68  case 2 : output += static\_cast<Uint8>(\*begin++); output <<= 6;

69  case 1 : output += static\_cast<Uint8>(\*begin++); output <<= 6;

70  case 0 : output += static\_cast<Uint8>(\*begin++);

71  }

72  output -= offsets[trailingBytes];

73  }

74  else

75  {

76  // Incomplete character

77  begin = end;

78  output = replacement;

79  }

80

81  return begin;

82 }

83

84

86 template <typename Out>

87 Out [Utf<8>::encode](http://docs.google.com/classUtf.htm)(Uint32 input, Out output, Uint8 replacement)

88 {

89  // Some useful precomputed data

90  static const Uint8 firstBytes[7] =

91  {

92  0x00, 0x00, 0xC0, 0xE0, 0xF0, 0xF8, 0xFC

93  };

94

95  // encode the character

96  if ((input > 0x0010FFFF) || ((input >= 0xD800) && (input <= 0xDBFF)))

97  {

98  // Invalid character

99  if (replacement)

100  \*output++ = replacement;

101  }

102  else

103  {

104  // Valid character

105

106  // Get the number of bytes to write

107  int bytestoWrite = 1;

108  if (input < 0x80) bytestoWrite = 1;

109  else if (input < 0x800) bytestoWrite = 2;

110  else if (input < 0x10000) bytestoWrite = 3;

111  else if (input <= 0x0010FFFF) bytestoWrite = 4;

112

113  // Extract the bytes to write

114  Uint8 bytes[4];

115  switch (bytestoWrite)

116  {

117  case 4 : bytes[3] = static\_cast<Uint8>((input | 0x80) & 0xBF); input >>= 6;

118  case 3 : bytes[2] = static\_cast<Uint8>((input | 0x80) & 0xBF); input >>= 6;

119  case 2 : bytes[1] = static\_cast<Uint8>((input | 0x80) & 0xBF); input >>= 6;

120  case 1 : bytes[0] = static\_cast<Uint8> (input | firstBytes[bytestoWrite]);

121  }

122

123  // Add them to the output

124  const Uint8\* currentByte = bytes;

125  switch (bytestoWrite)

126  {

127  case 4 : \*output++ = \*currentByte++;

128  case 3 : \*output++ = \*currentByte++;

129  case 2 : \*output++ = \*currentByte++;

130  case 1 : \*output++ = \*currentByte++;

131  }

132  }

133

134  return output;

135 }

136

137

139 template <typename In>

140 In [Utf<8>::next](http://docs.google.com/classUtf.htm)(In begin, In end)

141 {

142  Uint32 codepoint;

143  return decode(begin, end, codepoint);

144 }

145

146

148 template <typename In>

149 std::size\_t [Utf<8>::count](http://docs.google.com/classUtf.htm)(In begin, In end)

150 {

151  std::size\_t length = 0;

152  while (begin < end)

153  {

154  begin = next(begin, end);

155  ++length;

156  }

157

158  return length;

159 }

160

161

163 template <typename In, typename Out>

164 Out [Utf<8>::fromAnsi](http://docs.google.com/classUtf.htm)(In begin, In end, Out output, const std::locale& locale)

165 {

166  while (begin < end)

167  {

168  Uint32 codepoint = [Utf<32>::decodeAnsi](http://docs.google.com/classUtf.htm)(\*begin++, locale);

169  output = encode(codepoint, output);

170  }

171

172  return output;

173 }

174

175

177 template <typename In, typename Out>

178 Out [Utf<8>::fromWide](http://docs.google.com/classUtf.htm)(In begin, In end, Out output)

179 {

180  while (begin < end)

181  {

182  Uint32 codepoint = [Utf<32>::decodeWide](http://docs.google.com/classUtf.htm)(\*begin++);

183  output = encode(codepoint, output);

184  }

185

186  return output;

187 }

188

189

191 template <typename In, typename Out>

192 Out [Utf<8>::fromLatin1](http://docs.google.com/classUtf.htm)(In begin, In end, Out output)

193 {

194  // Latin-1 is directly compatible with Unicode encodings,

195  // and can thus be treated as (a sub-range of) UTF-32

196  while (begin < end)

197  output = encode(\*begin++, output);

198

199  return output;

200 }

201

202

204 template <typename In, typename Out>

205 Out [Utf<8>::toAnsi](http://docs.google.com/classUtf.htm)(In begin, In end, Out output, char replacement, const std::locale& locale)

206 {

207  while (begin < end)

208  {

209  Uint32 codepoint;

210  begin = decode(begin, end, codepoint);

211  output = [Utf<32>::encodeAnsi](http://docs.google.com/classUtf.htm)(codepoint, output, replacement, locale);

212  }

213

214  return output;

215 }

216

217

219 template <typename In, typename Out>

220 Out [Utf<8>::toWide](http://docs.google.com/classUtf.htm)(In begin, In end, Out output, wchar\_t replacement)

221 {

222  while (begin < end)

223  {

224  Uint32 codepoint;

225  begin = decode(begin, end, codepoint);

226  output = [Utf<32>::encodeWide](http://docs.google.com/classUtf.htm)(codepoint, output, replacement);

227  }

228

229  return output;

230 }

231

232

234 template <typename In, typename Out>

235 Out [Utf<8>::toLatin1](http://docs.google.com/classUtf.htm)(In begin, In end, Out output, char replacement)

236 {

237  // Latin-1 is directly compatible with Unicode encodings,

238  // and can thus be treated as (a sub-range of) UTF-32

239  while (begin < end)

240  {

241  Uint32 codepoint;

242  begin = decode(begin, end, codepoint);

243  \*output++ = codepoint < 256 ? static\_cast<char>(codepoint) : replacement;

244  }

245

246  return output;

247 }

248

249

251 template <typename In, typename Out>

252 Out [Utf<8>::toUtf8](http://docs.google.com/classUtf.htm)(In begin, In end, Out output)

253 {

254  while (begin < end)

255  \*output++ = \*begin++;

256

257  return output;

258 }

259

260

262 template <typename In, typename Out>

263 Out [Utf<8>::toUtf16](http://docs.google.com/classUtf.htm)(In begin, In end, Out output)

264 {

265  while (begin < end)

266  {

267  Uint32 codepoint;

268  begin = decode(begin, end, codepoint);

269  output = [Utf<16>::encode](http://docs.google.com/classUtf.htm)(codepoint, output);

270  }

271

272  return output;

273 }

274

275

277 template <typename In, typename Out>

278 Out [Utf<8>::toUtf32](http://docs.google.com/classUtf.htm)(In begin, In end, Out output)

279 {

280  while (begin < end)

281  {

282  Uint32 codepoint;

283  begin = decode(begin, end, codepoint);

284  \*output++ = codepoint;

285  }

286

287  return output;

288 }

289

290

292 template <typename In>

293 In [Utf<16>::decode](http://docs.google.com/classUtf.htm)(In begin, In end, Uint32& output, Uint32 replacement)

294 {

295  Uint16 first = \*begin++;

296

297  // If it's a surrogate pair, first convert to a single UTF-32 character

298  if ((first >= 0xD800) && (first <= 0xDBFF))

299  {

300  if (begin < end)

301  {

302  Uint32 second = \*begin++;

303  if ((second >= 0xDC00) && (second <= 0xDFFF))

304  {

305  // The second element is valid: convert the two elements to a UTF-32 character

306  output = static\_cast<Uint32>(((first - 0xD800) << 10) + (second - 0xDC00) + 0x0010000);

307  }

308  else

309  {

310  // Invalid character

311  output = replacement;

312  }

313  }

314  else

315  {

316  // Invalid character

317  begin = end;

318  output = replacement;

319  }

320  }

321  else

322  {

323  // We can make a direct copy

324  output = first;

325  }

326

327  return begin;

328 }

329

330

332 template <typename Out>

333 Out [Utf<16>::encode](http://docs.google.com/classUtf.htm)(Uint32 input, Out output, Uint16 replacement)

334 {

335  if (input < 0xFFFF)

336  {

337  // The character can be copied directly, we just need to check if it's in the valid range

338  if ((input >= 0xD800) && (input <= 0xDFFF))

339  {

340  // Invalid character (this range is reserved)

341  if (replacement)

342  \*output++ = replacement;

343  }

344  else

345  {

346  // Valid character directly convertible to a single UTF-16 character

347  \*output++ = static\_cast<Uint16>(input);

348  }

349  }

350  else if (input > 0x0010FFFF)

351  {

352  // Invalid character (greater than the maximum unicode value)

353  if (replacement)

354  \*output++ = replacement;

355  }

356  else

357  {

358  // The input character will be converted to two UTF-16 elements

359  input -= 0x0010000;

360  \*output++ = static\_cast<Uint16>((input >> 10) + 0xD800);

361  \*output++ = static\_cast<Uint16>((input & 0x3FFUL) + 0xDC00);

362  }

363

364  return output;

365 }

366

367

369 template <typename In>

370 In [Utf<16>::next](http://docs.google.com/classUtf.htm)(In begin, In end)

371 {

372  Uint32 codepoint;

373  return decode(begin, end, codepoint);

374 }

375

376

378 template <typename In>

379 std::size\_t [Utf<16>::count](http://docs.google.com/classUtf.htm)(In begin, In end)

380 {

381  std::size\_t length = 0;

382  while (begin < end)

383  {

384  begin = next(begin, end);

385  ++length;

386  }

387

388  return length;

389 }

390

391

393 template <typename In, typename Out>

394 Out [Utf<16>::fromAnsi](http://docs.google.com/classUtf.htm)(In begin, In end, Out output, const std::locale& locale)

395 {

396  while (begin < end)

397  {

398  Uint32 codepoint = [Utf<32>::decodeAnsi](http://docs.google.com/classUtf.htm)(\*begin++, locale);

399  output = encode(codepoint, output);

400  }

401

402  return output;

403 }

404

405

407 template <typename In, typename Out>

408 Out [Utf<16>::fromWide](http://docs.google.com/classUtf.htm)(In begin, In end, Out output)

409 {

410  while (begin < end)

411  {

412  Uint32 codepoint = [Utf<32>::decodeWide](http://docs.google.com/classUtf.htm)(\*begin++);

413  output = encode(codepoint, output);

414  }

415

416  return output;

417 }

418

419

421 template <typename In, typename Out>

422 Out [Utf<16>::fromLatin1](http://docs.google.com/classUtf.htm)(In begin, In end, Out output)

423 {

424  // Latin-1 is directly compatible with Unicode encodings,

425  // and can thus be treated as (a sub-range of) UTF-32

426  while (begin < end)

427  \*output++ = \*begin++;

428

429  return output;

430 }

431

432

434 template <typename In, typename Out>

435 Out [Utf<16>::toAnsi](http://docs.google.com/classUtf.htm)(In begin, In end, Out output, char replacement, const std::locale& locale)

436 {

437  while (begin < end)

438  {

439  Uint32 codepoint;

440  begin = decode(begin, end, codepoint);

441  output = [Utf<32>::encodeAnsi](http://docs.google.com/classUtf.htm)(codepoint, output, replacement, locale);

442  }

443

444  return output;

445 }

446

447

449 template <typename In, typename Out>

450 Out [Utf<16>::toWide](http://docs.google.com/classUtf.htm)(In begin, In end, Out output, wchar\_t replacement)

451 {

452  while (begin < end)

453  {

454  Uint32 codepoint;

455  begin = decode(begin, end, codepoint);

456  output = [Utf<32>::encodeWide](http://docs.google.com/classUtf.htm)(codepoint, output, replacement);

457  }

458

459  return output;

460 }

461

462

464 template <typename In, typename Out>

465 Out [Utf<16>::toLatin1](http://docs.google.com/classUtf.htm)(In begin, In end, Out output, char replacement)

466 {

467  // Latin-1 is directly compatible with Unicode encodings,

468  // and can thus be treated as (a sub-range of) UTF-32

469  while (begin < end)

470  {

471  \*output++ = \*begin < 256 ? static\_cast<char>(\*begin) : replacement;

472  begin++;

473  }

474

475  return output;

476 }

477

478

480 template <typename In, typename Out>

481 Out [Utf<16>::toUtf8](http://docs.google.com/classUtf.htm)(In begin, In end, Out output)

482 {

483  while (begin < end)

484  {

485  Uint32 codepoint;

486  begin = decode(begin, end, codepoint);

487  output = [Utf<8>::encode](http://docs.google.com/classUtf.htm)(codepoint, output);

488  }

489

490  return output;

491 }

492

493

495 template <typename In, typename Out>

496 Out [Utf<16>::toUtf16](http://docs.google.com/classUtf.htm)(In begin, In end, Out output)

497 {

498  while (begin < end)

499  \*output++ = \*begin++;

500

501  return output;

502 }

503

504

506 template <typename In, typename Out>

507 Out [Utf<16>::toUtf32](http://docs.google.com/classUtf.htm)(In begin, In end, Out output)

508 {

509  while (begin < end)

510  {

511  Uint32 codepoint;

512  begin = decode(begin, end, codepoint);

513  \*output++ = codepoint;

514  }

515

516  return output;

517 }

518

519

521 template <typename In>

522 In [Utf<32>::decode](http://docs.google.com/classUtf.htm)(In begin, In /\*end\*/, Uint32& output, Uint32 /\*replacement\*/)

523 {

524  output = \*begin++;

525  return begin;

526 }

527

528

530 template <typename Out>

531 Out [Utf<32>::encode](http://docs.google.com/classUtf.htm)(Uint32 input, Out output, Uint32 /\*replacement\*/)

532 {

533  \*output++ = input;

534  return output;

535 }

536

537

539 template <typename In>

540 In [Utf<32>::next](http://docs.google.com/classUtf.htm)(In begin, In /\*end\*/)

541 {

542  return ++begin;

543 }

544

545

547 template <typename In>

548 std::size\_t [Utf<32>::count](http://docs.google.com/classUtf.htm)(In begin, In end)

549 {

550  return begin - end;

551 }

552

553

555 template <typename In, typename Out>

556 Out [Utf<32>::fromAnsi](http://docs.google.com/classUtf.htm)(In begin, In end, Out output, const std::locale& locale)

557 {

558  while (begin < end)

559  \*output++ = decodeAnsi(\*begin++, locale);

560

561  return output;

562 }

563

564

566 template <typename In, typename Out>

567 Out [Utf<32>::fromWide](http://docs.google.com/classUtf.htm)(In begin, In end, Out output)

568 {

569  while (begin < end)

570  \*output++ = decodeWide(\*begin++);

571

572  return output;

573 }

574

575

577 template <typename In, typename Out>

578 Out [Utf<32>::fromLatin1](http://docs.google.com/classUtf.htm)(In begin, In end, Out output)

579 {

580  // Latin-1 is directly compatible with Unicode encodings,

581  // and can thus be treated as (a sub-range of) UTF-32

582  while (begin < end)

583  \*output++ = \*begin++;

584

585  return output;

586 }

587

588

590 template <typename In, typename Out>

591 Out [Utf<32>::toAnsi](http://docs.google.com/classUtf.htm)(In begin, In end, Out output, char replacement, const std::locale& locale)

592 {

593  while (begin < end)

594  output = encodeAnsi(\*begin++, output, replacement, locale);

595

596  return output;

597 }

598

599

601 template <typename In, typename Out>

602 Out [Utf<32>::toWide](http://docs.google.com/classUtf.htm)(In begin, In end, Out output, wchar\_t replacement)

603 {

604  while (begin < end)

605  output = encodeWide(\*begin++, output, replacement);

606

607  return output;

608 }

609

610

612 template <typename In, typename Out>

613 Out [Utf<32>::toLatin1](http://docs.google.com/classUtf.htm)(In begin, In end, Out output, char replacement)

614 {

615  // Latin-1 is directly compatible with Unicode encodings,

616  // and can thus be treated as (a sub-range of) UTF-32

617  while (begin < end)

618  {

619  \*output++ = \*begin < 256 ? static\_cast<char>(\*begin) : replacement;

620  begin++;

621  }

622

623  return output;

624 }

625

626

628 template <typename In, typename Out>

629 Out [Utf<32>::toUtf8](http://docs.google.com/classUtf.htm)(In begin, In end, Out output)

630 {

631  while (begin < end)

632  output = [Utf<8>::encode](http://docs.google.com/classUtf.htm)(\*begin++, output);

633

634  return output;

635 }

636

638 template <typename In, typename Out>

639 Out [Utf<32>::toUtf16](http://docs.google.com/classUtf.htm)(In begin, In end, Out output)

640 {

641  while (begin < end)

642  output = [Utf<16>::encode](http://docs.google.com/classUtf.htm)(\*begin++, output);

643

644  return output;

645 }

646

647

649 template <typename In, typename Out>

650 Out [Utf<32>::toUtf32](http://docs.google.com/classUtf.htm)(In begin, In end, Out output)

651 {

652  while (begin < end)

653  \*output++ = \*begin++;

654

655  return output;

656 }

657

658

660 template <typename In>

661 Uint32 [Utf<32>::decodeAnsi](http://docs.google.com/classUtf.htm)(In input, const std::locale& locale)

662 {

663  // On Windows, gcc's standard library (glibc++) has almost

664  // no support for Unicode stuff. As a consequence, in this

665  // context we can only use the default locale and ignore

666  // the one passed as parameter.

667

668  #if defined(SFML\_SYSTEM\_WINDOWS) && /\* if Windows ... \*/ \

669  (defined(\_\_GLIBCPP\_\_) || defined (\_\_GLIBCXX\_\_)) && /\* ... and standard library is glibc++ ... \*/ \

670  !(defined(\_\_SGI\_STL\_PORT) || defined(\_STLPORT\_VERSION)) /\* ... and STLPort is not used on top of it \*/

671

672  (void)locale; // to avoid warnings

673

674  wchar\_t character = 0;

675  mbtowc(&character, &input, 1);

676  return static\_cast<Uint32>(character);

677

678  #else

679

680  // Get the facet of the locale which deals with character conversion

681  const std::ctype<wchar\_t>& facet = std::use\_facet< std::ctype<wchar\_t> >(locale);

682

683  // Use the facet to convert each character of the input string

684  return static\_cast<Uint32>(facet.widen(input));

685

686  #endif

687 }

688

689

691 template <typename In>

692 Uint32 [Utf<32>::decodeWide](http://docs.google.com/classUtf.htm)(In input)

693 {

694  // The encoding of wide characters is not well defined and is left to the system;

695  // however we can safely assume that it is UCS-2 on Windows and

696  // UCS-4 on Unix systems.

697  // In both cases, a simple copy is enough (UCS-2 is a subset of UCS-4,

698  // and UCS-4 \*is\* UTF-32).

699

700  return input;

701 }

702

703

705 template <typename Out>

706 Out [Utf<32>::encodeAnsi](http://docs.google.com/classUtf.htm)(Uint32 codepoint, Out output, char replacement, const std::locale& locale)

707 {

708  // On Windows, gcc's standard library (glibc++) has almost

709  // no support for Unicode stuff. As a consequence, in this

710  // context we can only use the default locale and ignore

711  // the one passed as parameter.

712

713  #if defined(SFML\_SYSTEM\_WINDOWS) && /\* if Windows ... \*/ \

714  (defined(\_\_GLIBCPP\_\_) || defined (\_\_GLIBCXX\_\_)) && /\* ... and standard library is glibc++ ... \*/ \

715  !(defined(\_\_SGI\_STL\_PORT) || defined(\_STLPORT\_VERSION)) /\* ... and STLPort is not used on top of it \*/

716

717  (void)locale; // to avoid warnings

718

719  char character = 0;

720  if (wctomb(&character, static\_cast<wchar\_t>(codepoint)) >= 0)

721  \*output++ = character;

722  else if (replacement)

723  \*output++ = replacement;

724

725  return output;

726

727  #else

728

729  // Get the facet of the locale which deals with character conversion

730  const std::ctype<wchar\_t>& facet = std::use\_facet< std::ctype<wchar\_t> >(locale);

731

732  // Use the facet to convert each character of the input string

733  \*output++ = facet.narrow(static\_cast<wchar\_t>(codepoint), replacement);

734

735  return output;

736

737  #endif

738 }

739

740

742 template <typename Out>

743 Out [Utf<32>::encodeWide](http://docs.google.com/classUtf.htm)(Uint32 codepoint, Out output, wchar\_t replacement)

744 {

745  // The encoding of wide characters is not well defined and is left to the system;

746  // however we can safely assume that it is UCS-2 on Windows and

747  // UCS-4 on Unix systems.

748  // For UCS-2 we need to check if the source characters fits in (UCS-2 is a subset of UCS-4).

749  // For UCS-4 we can do a direct copy (UCS-4 \*is\* UTF-32).

750

751  switch (sizeof(wchar\_t))

752  {

753  case 4:

754  {

755  \*output++ = static\_cast<wchar\_t>(codepoint);

756  break;

757  }

758

759  default:

760  {

761  if ((codepoint <= 0xFFFF) && ((codepoint < 0xD800) || (codepoint > 0xDFFF)))

762  {

763  \*output++ = static\_cast<wchar\_t>(codepoint);

764  }

765  else if (replacement)

766  {

767  \*output++ = replacement;

768  }

769  break;

770  }

771  }

772

773  return output;

774 }

Copyright � Laurent Gomila  ::  Documentation generated by [doxygen](http://www.doxygen.org/)  ::