NXP Semiconductors

Document identifier: eTPURDCCUG

Rev. 0, March 2020

eTPU RDC and RDC Checker User Guide

Contents

Chapter 1
Product Description

This user guide provides detailed description of eTPU RDC function implementation as well as host RDC Checker. The user guide provides complete guidance for configurations and usage of both SW products to ensure safety operation. The SW described within this manual is applicable to devices containing Enhanced Timer Processing Unit (eTPU, eTPU2 and eTPU2+).

Acronyms and terms used in the document are listed below.

Table 1. Acronym and definition

Acronym	Definition
RDC	Resolver to Digital Converter
eTPU	Enhanced Time Processing Unit
SDADC	Sigma-Delta Analog to Digital Converter
ATO	Angle-Tracking Observer
SRF	Safety Runtime Framework
API	Application Interface
PIT	Periodic Interrupt Timer
HSR	Host Service Request
ISR	Interrupt Service Request
PWM	Pulse Width Modulation

1.1 Resolver Digital Interface (RESOLVER)

The Resolver Digital Interface eTPU function (RESOLVER) uses one eTPU channel to generate a 50% duty-cycle PWM output signal to be passed through an external low-pass filter and used as a resolver excitation signal. In the resolver position sensor, this excitation signal is modulated by sine and cosine of the actual motor angle. The feedback Sine and Cosine signals are sampled by an on-chip ADC and the conversion results can be transferred to eTPU DATA RAM by eDMA. eTPU function RESOLVER can then process the digital samples of resolver output signals. Motor angular position, angular speed, a revolution counter, and diagnostics are results of the Sine and Cosine feedback signal processing.

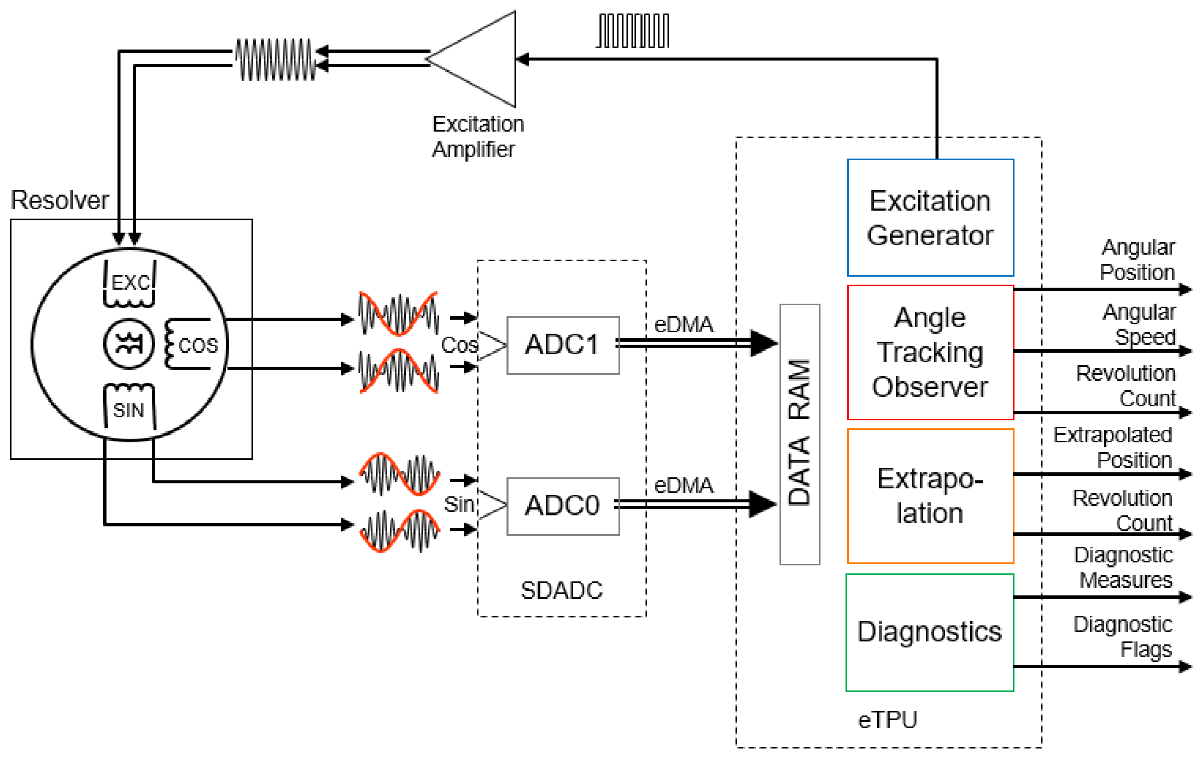


Figure 1. Resolver Digital Interface block diagram

Processing of the feedback signals is executed on a separate channel. Another channel is used to perform linear extrapolation of the last updated position from ATO to any other time. Optionally, another eTPU channel can be used to process diagnostics either on the same eTPU engine after the feedback signal processing is finished (see the channel assignment example in Table 3) or on the other eTPU engine in parallel to the motor angle and speed calculation, see Table 4. This enables the CPU application to read the new motor angle and, at the same time, check the diagnostic results to ensure the motor angle is correct.

As an alternative to diagnostics running on eTPU Resolver function there is the RDC Checker, an external checker running on CPU. This option allows eTPU Resolver to be a part of ASIL-D decomposition, where the eTPU Resolver function (QM(D)) runs in Non-Trusted Environment (NTE) and the RDC Checker (ASIL-D(D)) in Trusted Environment (TE)

Features:

- Generation of excitation signal – a 50% PWM wave to be filtered externally.
- Adaptive phase control of the excitation signal.
- Motor angle and speed tracking
- Extrapolation of the resulting angle to a defined time-position
- Diagnostic measurements (optionally running in parallel on the other eTPU engine)
- Diagnostic flag settings (optionally running in parallel on the other eTPU engine)

1.1.1 Excitation signal generation and adaptive phase control

The Resolver Digital Interface eTPU function generates a square output wave of a defined period which is intended to be filtered and gained by an external circuitry to get the excitation sine wave. An example of the external circuitry schematic for filtration of

10 kHz output square signal to gain sine-wave resolver excitation of the same frequency is shown in the following figure. Note that this is just a simplified example, not all the connections are shown.

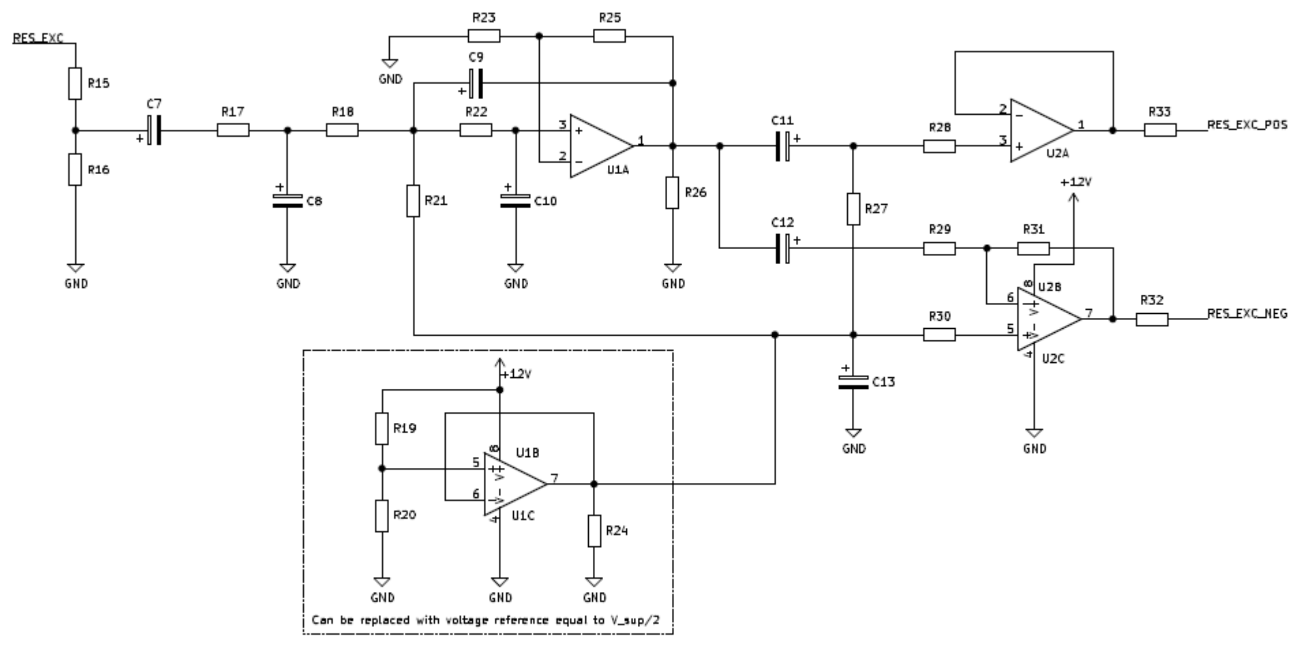


Figure 2. An example schematic of the excitation generation circuitry

The adaptive control of the excitation signal phase is enabled by setting the bit FS_ETPU_RESOLVER_OPTIONS_EXC_ADAPTATION_ON in *resolver_config.options*. Using a PI controller (*exc_p_gain*, *exc_i_gain*), the excitation signal phase is adjusted so, that the zero crossings of the Sine and Cosine feedback signals are at the required position and the signals are phase-aligned in the input signal buffers, see the following figure.

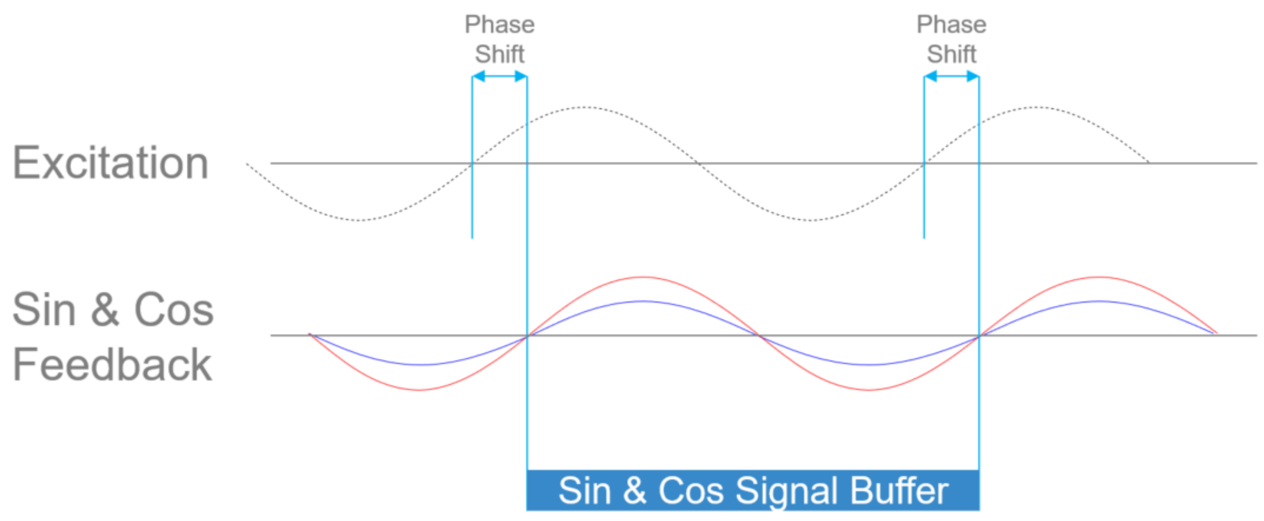


Figure 3. Adaptive phase shift and feedback signals alignment

1.1.2 Sine and Cosine feedback signal digitization

The Sine and Cosine analog feedback signals need to be converted to a digital representation and transferred to eTPU DATA RAM. This should be done independently of the CPU using an on-chip ADC and eDMA. Although any of the ADC modules can be used, the described configuration adopts the Sigma-Delta ADC (SDADC).

Differential ADC inputs are used, which simplifies the input circuitry. The following figure shows an example of the external circuitry schematic.

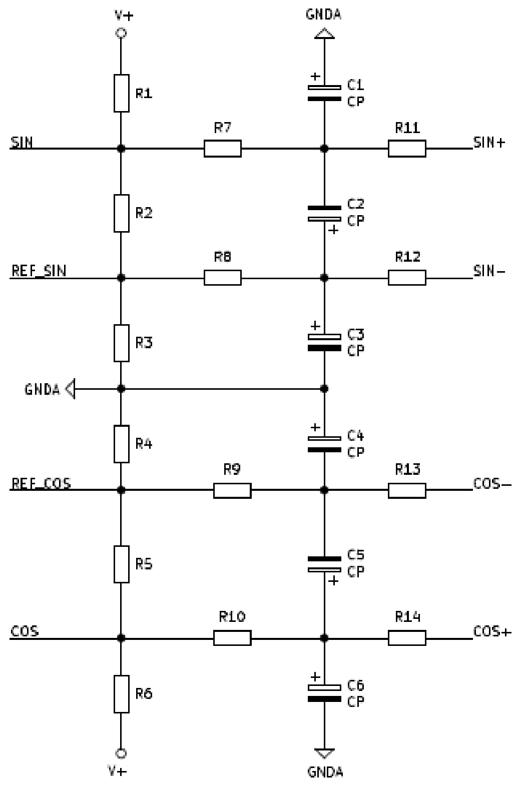


Figure 4. An example schematic of the resolver Sine and Cosine feedback signal circuitry

Two SDADC modules are used to continuously sample the Sine and Cosine signals in parallel. They are configured to obtain 32 samples of each signal per period. The following table shows the configuration for a 10 kHz excitation signal and a 320 kHz sampling frequency in detail.

Table 2. SDADC Configuration.

Configuration Item	Value
SDADC clock	200 MHz / 13 = 15.38 MHz (available range 4 – 16 MHz)
ADC decimation rate	24
Resulting output data rate	200 MHz / 13 / (2 * 24) = 320,512.8 Hz
Input mode	differential (recommended)
High-pass filter	enabled

Table continues on the next page...

Table 2. SDADC Configuration. (continued)

Configuration Item	Value
FIFO size	16 words
FIFO threshold	16 words
DMA request on FIFO full	selected and enabled

The ADC generates a DMA request every time when 16 samples are ready in the FIFO. These 16 samples correspond to either the first or the second half-period of the input signal. The DMA transfers those samples to the eTPU DATA RAM, either to the first or the second half of the signal buffer, and subsequently evokes the eTPU to process the new data. This is done by linking another DMA channel which transfers a constant value to the eTPU Host Service Request (HSR) register. Then eTPU processes the HSR. Different HSRs are used for the first and for the second half-period:

- FS_ETPU_RESOLVER_HSR_UPDATE_1ST
- FS_ETPU_RESOLVER_HSR_UPDATE_2ND

Altogether, three DMA channels are used – two channels to transfer the ADC data of Sine and Cosine signals, and one channel to transfer the HSRs. The following table shows eDMA configuration in detail.

Table 3. eDMA Configuration

Configuration Item	Sine ADC FIFO DMA channel	Cosine ADC FIFO DMA channel	Linked HSR DMA channel
Source address	&SDADC_x.CDR.R	&SDADC_y.CDR.R	&link_const[0] const uint32_t link_const[] = {FS_ETPU_RESOLVER_HSR_UPDATE_1ST, FS_ETPU_RESOLVER_HSR_UPDATE_2ND};
Destination address	resolver_instance signals_pba	resolver_instance signals_pba + 64	&ETPU.CHAN[resolver_instance.chan_num_exc].HSRR.R
Source transfer size / modulo	32-bits / 0 bytes	32-bits / 0 bytes	32-bits / 0 bytes
Destination transfer size / modulo	32-bits / 64 bytes	32-bits / 64 bytes	32-bits / 0 bytes
Source address offset	0 bytes	0 bytes	4 bytes
Destination address offset	4 bytes	4 bytes	0 bytes
Minor loop byte count	64 bytes	64 bytes	4 bytes
Major loop iteration count	2	2	2
Last source address adjustment	0 bytes	0 bytes	-8 bytes
Last destination address adjustment	-128 bytes	-128 bytes	0 bytes

Table continues on the next page...

Table 3. eDMA Configuration (continued)

Configuration Item	Sine ADC FIFO DMA channel	Cosine ADC FIFO DMA channel	Linked HSR DMA channel
Channel to channel linking	disabled	enabled	disabled
Linked channel	-	HSR DMA channel	-

1.1.3 Resolver angle and speed tracking

Having the Sine and Cosine feedback signals in the signal buffer, the actual resolver angle could theoretically be calculated using the arcustangent function after signal demodulations.

$$\theta = \arctan\left(\frac{\sin(\theta)}{\cos(\theta)}\right)$$

where sin(θ) and cos(θ) are the demodulated resolver feedback signals.

This straightforward way has several disadvantages, as a long calculation time or a low noise immunity. Due to those reasons the Angle Tracking Observer (ATO) is introduced. The ATO tracks angular motor speed and calculates actual motor angle using a prediction and correction approach. The following figure shows the block diagram of this algorithm.

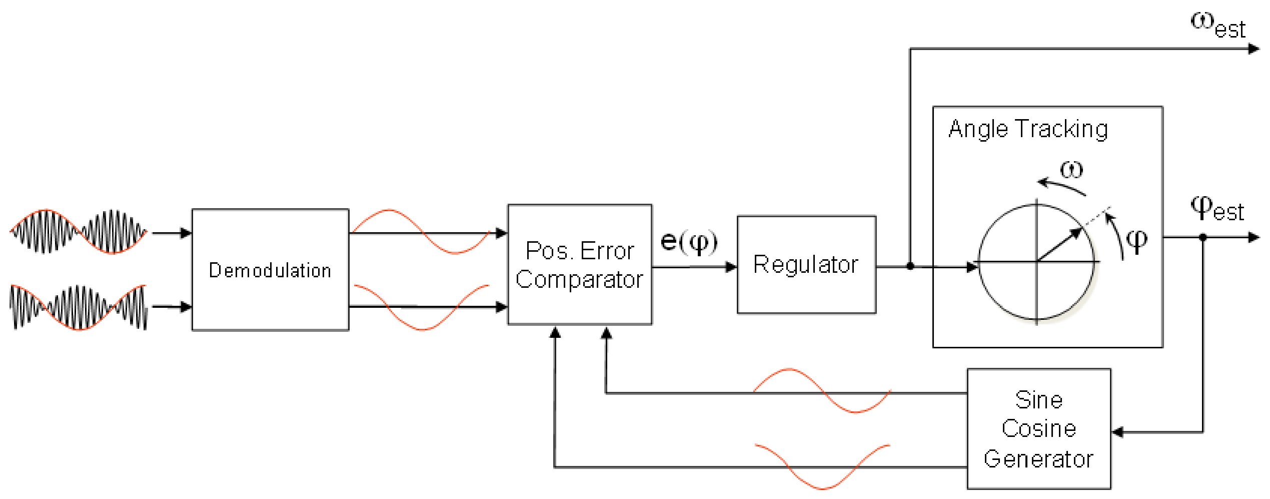


Figure 5. Angle tracking observer

The ATO consists of the following blocks:

- Demodulation: The excitation signal is modulated by Sine and Cosine of the motor angle by the resolver. The Demodulation block extracts the modulation signal out of the excitation (carrying) signal. This is done by oversampling the modulated excitation signal (sampling rate 320.5128 kHz) and thus obtaining 32 samples per period, 16 samples per half-period respectively. The half period is being demodulated by factor of the carrier signal (sine) and median filter is used to determine the modulation signal amplitude (resolver angle sine and cosine). Median filter is suppressing the impulse noise.
- Position Error Comparator: This block takes the Sine and Cosine of the motor angle measured from the resolver feedback signals and the Sine and Cosine calculated from the motor angle estimated by the ATO algorithm. Using those inputs, an angle error is calculated. The calculation is based on the trigonometric identity.

  sin(α - β) = sin (α)cos(β) - cos(α)sin(β)

For small angles following simplification can be applied.

$\sin(\mathcal{E}) = \cong \mathcal{E}$

For bigger angles a convergency of the ATO algorithm can be proofed. As a result, the position error is calculated.

$e(\theta) = \theta - \theta_{est} \cong \sin(\theta)\cos(\theta_{est}) - \cos(\theta)\sin(\theta_{est})$

Where $\sin(\theta)$ and $\cos(\theta)$ are the demodulated resolver feedback signals, $\cos(\theta_{est})$ and $\sin(\theta_{est})$ are calculated sine and cosine values of the ATO motor angle output, and is the resulting motor angle error.

- Regulator: The Regulator takes the position error as an input and returns estimated motor speed as an output. A 24-bit fractional PI controller is implemented for this task. The Excel file Resolver ATO PI-gains.xlsx, which is included in the attached software package, might help to set the PI controller gains.
- Angle Tracking: The Angle Tracking block adjusts the actual motor angle using the updated motor speed. As a result, the estimated motor angle is returned.

There are two Angle Tracking Observer updates per period, separately using the first and the second half-period of the Sine and Cosine signals.

1.1.4 Extrapolation

The calculated motor angle corresponds to the time position of the half-period center. In a motor control application, the motor angle reading should correspond in time to the phase current readings. To achieve this, there is a feature to linearly extrapolate the calculated motor angle to another time position, using the actual motor speed.

The angle and speed are extrapolated to the time when the extrapolation is triggered. It can be triggered by:

- Host service request from application
- Link from other eTPU channel
- Input transition detection (rising/falling edge)

To enable calculation of the extrapolated angle values the FS_ETPU_RESOLVER_OPTIONS_EXTRAPOLATION_ON has to be configured in *resolver_config.options*, The API function *fs_etpu_resolver_get_outputs_extrapolated()* can be called to read the extrapolated results.

1.1.5 Diagnostics

There are two ways how the operation of the eTPU Resolver can be checked. The first one is to run diagnostics on eTPU as described in eTPU diagnostics. The second option is to use RDC (Resolver-to-Digital-Converter) checker, see description in RDC checker.

1.1.5.1 eTPU diagnostics

The Resolver Diagnostics on eTPU can run in parallel to the motor position processing on the other eTPU engine or after the motor position processing on the same engine. See the following tables.

Table 4. RESOLVER channel assignment – Example 1

Channel	Assignment	Note
chan_num_exc	9	Excitation signal generation runs on eTPU A channel 9.
chan_num_ATO	8	Motor position processing runs on eTPU A channel 8.
chan_num_diag	10	Diagnostics run **after** motor position processing on eTPU A channel 10.

Table 5. RESOLVER channel assignment – Example 2

Channel	Assignment	Note
chan_num_exc	9	Excitation signal generation runs on eTPU A channel 9.
chan_num_ATO	8	Motor position processing runs on eTPU A channel 8.
chan_num_diag	3+64	Diagnostics run **in parallel** to motor position processing on eTPU B channel 3.

The Resolver Diagnostics are enabled by setting following bits in *resolver_config.options*

- FS_ETPU_RESOLVER_OPTIONS_DIAG_MEASURES_ON
- FS_ETPU_RESOLVER_OPTIONS_DIAG_FLAGS_ON.

The diagnostics processing is handled partly by the eTPU function and partly by the CPU API. On the eTPU site, the Sine and Cosine feedback signals are diagnosed in two phases:

- Diagnostic Measurements - signal properties are measured
- Diagnostic Flags - flags are set upon comparing the measured values against the thresholds

On the CPU site, additional flag decoding can be done. This part is somewhat dependent on the external circuitry (excitation amplifier, input filters) and that is why this part is done within API, so that the user can easily modify the code according to the need.

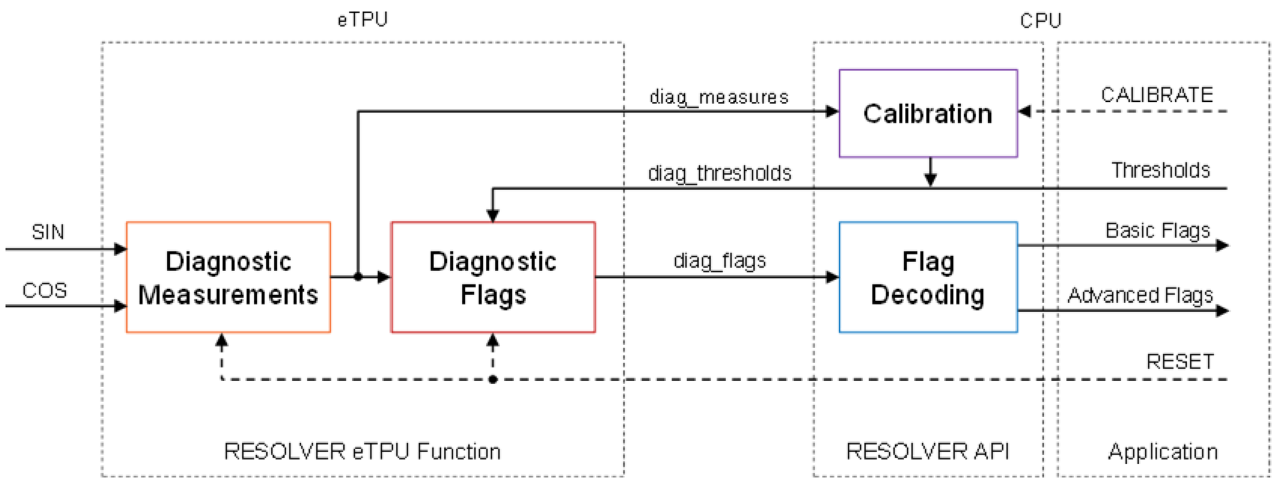
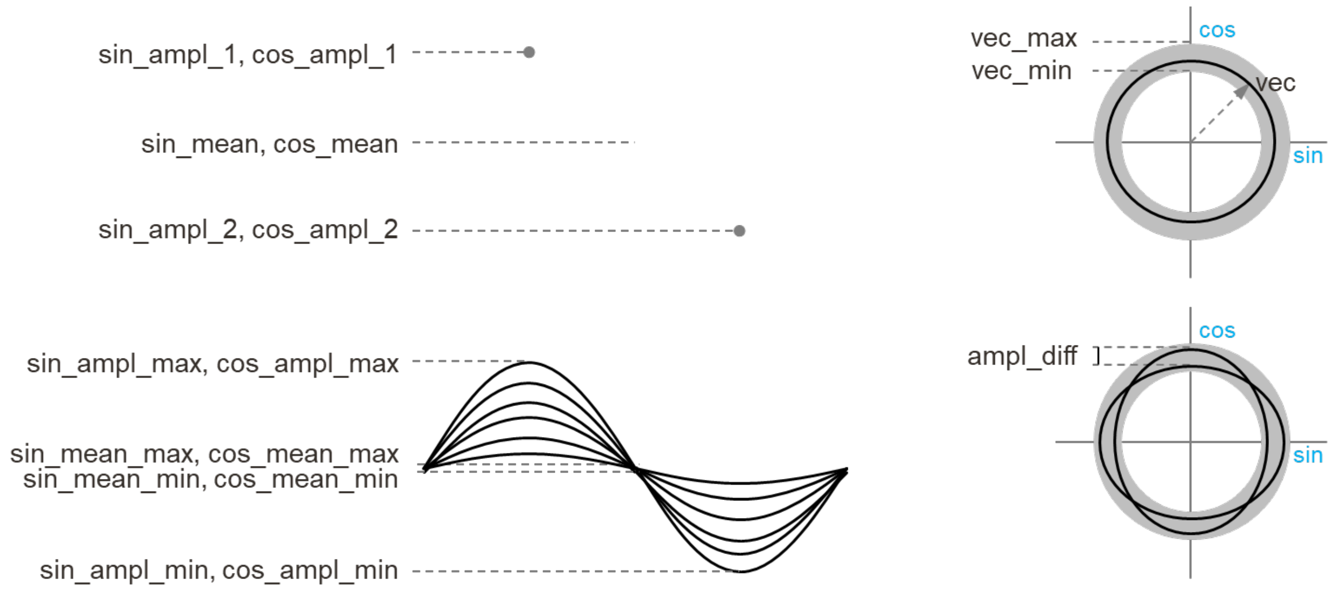


Figure 6. Resolver diagnostics overview

The diagnostic thresholds can be provided by the CPU application as manually configured constants, or thresholds can be determined from the measured signal properties by a call of *fs_etpu_resolver_diag_calibrate()* API function. Passing at least one full motor revolution guaranties enough data for that automatic calibration. The application also needs to make sure the input signals are not corrupted before calling the calibration.

Both the diagnostic measurements and diagnostic flags include some actual and some cumulative values. In order to reset and restart the measurement and the flag setting, the API function *fs_etpu_resolver_diag_reset()* can be called. The diagnostic measures are shown in the following figure and RESOLVER diagnostic flags lists all the values which are computed by the diagnostic measurements.

Figure 7. Resolver diagnostic measures


sin_ampl_1, cos_ampl_1
sin_mean, cos_mean
sin_ampl_2, cos_ampl_2
sin_ampl_max, cos_ampl_max
sin_mean_max, cos_mean_max
sin_mean_min, cos_mean_min
sin_ampl_min, cos_ampl_min
vec_max
vec_min
cos
vec
sin
ampl_diff
cos
sin

NOTE

The resolver feedback signals have to be conditioned so that the amplitudes being displayed on the unitary circle of both sine and cosine are within the range from ±0.3 to ±0.8. The upper limit should not be exceeded in case diagnostics are planned to be used to allow the excursion of the signals to be detected when the unitary circle is distorted due to shortcuts and signal losses.

Table 6. RESOLVER diagnostic measures

resolver_diag_measures_t structure member		Description
sin_ampl_1	cos_ampl_1	Amplitude of the first, positive, half period
sin_ampl_2	cos_ampl_2	Amplitude of the second, negative, half period
sin_ampl_min	cos_ampl_min	The lowest amplitude ever detected
sin_ampl_max	cos_ampl_max	The highest amplitude ever detected
sin_mean	cos_mean	The signal mean value (DC component)
sin_mean_min	cos_mean_min	The lowest mean value ever detected
sin_mean_max	cos_mean_max	The highest mean value ever detected
vec		Square value of a vector formed by sin and cos amplitudes as its orthogonal components
vec_min		The lowest value of the square vector value ever detected
vec_max		The highest value of the square vector value ever detected
ampl_diff		The difference between the highest amplitude ever detected of one signal and the lowest amplitude ever detected of the other signal – unit circle deformation

The following table lists all the thresholds against which the diagnostic measures are compared.

Table 7. RESOLVER diagnostic thresholds

resolver_diag_thresholds_t structure member	Description
ampl_thrs	The highest absolute amplitude value threshold
mean_thrs	The highest absolute mean value threshold
vec_min_thrs	The lowest square vector threshold
vec_max_thrs	The highest square vector threshold
ampl_diff_thrs	The highest sin/cos amplitude difference threshold
ato_angle_err_thrs	The highest ATO angle error threshold
ato_speed_thrs	The highest ATO speed threshold

The following table lists all diagnostic flags and conditions under which the flags are set.

Table 8. RESOLVER diagnostic flags

resolver_diag_flags_t structure member bits		Condition	Description
SIN_AMPL	COS_AMPL	|ampl_1/2| > ampl_thrs	The amplitude is higher than the threshold
SIN_MEAN	COS_MEAN	|mean| > mean_thrs	The signal mean value is higher than the threshold
VEC_MIN		vec < vec_min_thrs	The square vector is lower than the threshold
VEC_MAX		vec > vec_max_thrs	The square vector is higher than the threshold
AMPL_DIFF		ampl_diff > ampl_diff_thrs	The sin/cos amplitude difference is higher than the threshold
ATO_ANGLE_ERR		|ato.angle_err| > ato_angle_err_thrs	The ATO angle error is higher than the threshold
ATO_SPEED		|ato.speed| > ato_speed_thrs	The ATO speed is higher than the threshold

The eTPU runs the diagnostics every time when the new inputs are available (on every update). The CPU can run the flag decoding on a lower frequency, based on the application needs. No diagnostic flag can be missed, because any time a flag is detected, it remains to be set within the cumulative value.

Table 9. RESOLVER actual and cumulative diagnostic flags

resolver_diag_flags_t structure member	Description
actual	Field of flags detected on the last update from the last diagnostic_measures data.

Table continues on the next page...

Table 9. RESOLVER actual and cumulative diagnostic flags (continued)

resolver_diag_flags_t structure member	Description
cumulative	Field of cumulated flags. The actual flags are OR-ed to the cumulative flags on every update. The cumulative flags are cleared by reset of diagnostics.

The CPU can take the diagnostic flags provided by the eTPU function and use them to decode a higher level of diagnostics, Basic diagnostic flags and Advanced diagnostic flags. The following tables list those flags and describes the flag setting conditions. Note that these conditions might not be generally applicable for all types of external hardware circuitry. You can easily modify the flag setting condition in the API code according to the application specifics.

Table 10. Basic diagnostic flags

resolver_diag_flags_basic bits	Condition	Description
LOS	VEC_MIN	Loss of Signal
DOS	VEC_MAX or AMPL_DIFF	Degradation of Signal
LOT	ATO_ANGLE_ERR or ATO_SPEED	Loss of Tracking

Table 11. Advanced diagnostic flags

resolver_diag_flags_ advanced bits	Condition – decoded in this order:	Description
SIN_DISCONNECT	SIN_MEAN and AMPL_DIFF and ATO_ANGLE_ERR	A wire of the SIN signal is disconnected.
COS_DISCONNECT	COS_MEAN and AMPL_DIFF and ATO_ANGLE_ERR	A wire of the COS signal is disconnected.
EXC_DISCONNECT	VEC_MIN and ATO_SPEED	A wire of the EXC signal is disconnected.
SIN_SHORT	SIN_MEAN and ATO_ANGLE_ERR	A shortcut on the SIN signal wire.
COS_SHORT	COS_MEAN and ATO_ANGLE_ERR	A shortcut on the COS signal wire.
EXC_SHORT	VEC_MIN	A shortcut on the EXC signal wire.

1.2 RDC checker

RDC checker provides an alternative way of detecting eTPU Resolver functionality or input signals failure. The key difference is that the diagnostics are moved to CPU. eTPU Resolver function optionally records data for the external (meaning external to eTPU) checker. This functionality is enabled by setting following bits in *resolver_config.options*:

- FS_ETPU_RESOLVER_OPTIONS_RDC_CHECKER_ON

This option enables to Resolver function to log following data into a structure which is then transferred by DMA into a system RAM, where data are taken care of by RDC Checker. The structure of logged data at eTPU side is as follows:

- `int24_t sin_8; /* Sine sample on 8th position from signal buffer corresponding to peak of 1st half period*/`
- `int24_t sin_24; /* Sine sample on 24th position from signal buffer corresponding to peak of 2nd half period*/`

- `int24_t cos_8; /* Cosine sample on 8th position from signal buffer corresponding to peak of 1st half period*/`
- `int24_t cos_24; /* Cosine sample on 24th position from signal buffer corresponding to peak of 2nd half period*/`
- `int24_t sin_16; /* Sine sample on 16th position from signal buffer corresponding to zero crossing*/`
- `int24_t cos_16; /* Cosine sample on 16th position from signal buffer corresponding to zero crossing*/`
- `fract24_t ATO_angle_8; /* ATO angle from 1st update*/`
- `int24_t timestamp_8; /* Timestamp for 1st update*/`
- `fract24_t ATO_angle_24; /* ATO angle from 2nd update */`
- `int24_t timestamp_24; /* Timestamp for 2nd update*/`
- `fract24_t extrapolated_angle; /* Last Extrapolated angle value */`

The block diagram of the RDC Checker and eTPU can be seen in the following figure. The RDC Checker can run either as a standalone checker or as a part of Safety Runtime Framework.

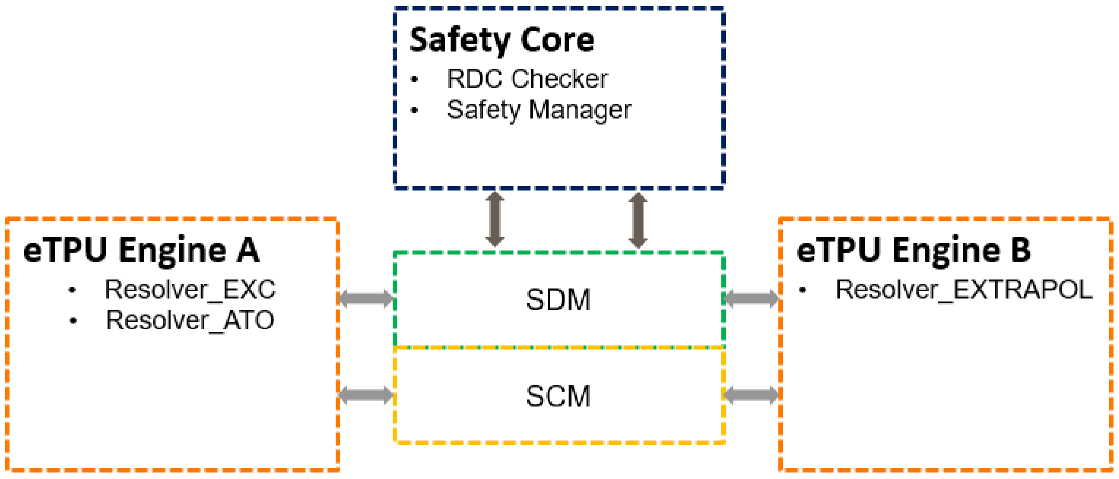


Figure 8. RDC checker and eTPU block diagram

RDC Checker consists of four main parts, the detailed block diagram can be seen in the following figure:

- Input signal checker
- ATO checker
- Extrapolation checker
- Timing checker

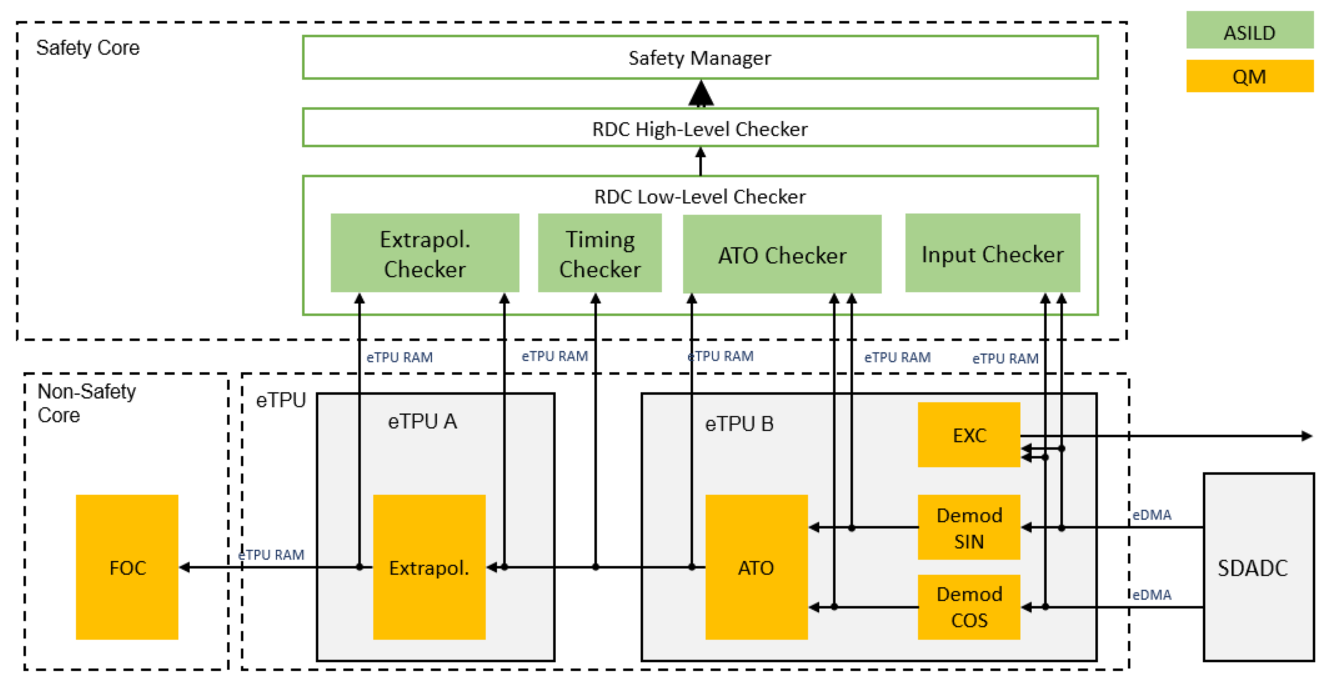


Figure 9. RDC checker detailed diagram

1.2.1 Input signal checker

Input signal checker inspects the Resolver feedback signals that are to be processed by eTPU Resolver function. It checks the samples that are assumed to be the maximum amplitude points: sin[8], cos[8], sin[24] and cos [24] out of 32 point sample buffers. Also for the purpose of phase shift checks the samples of assumed zero crossings: sin[16] and cos[16]. See the following figure.

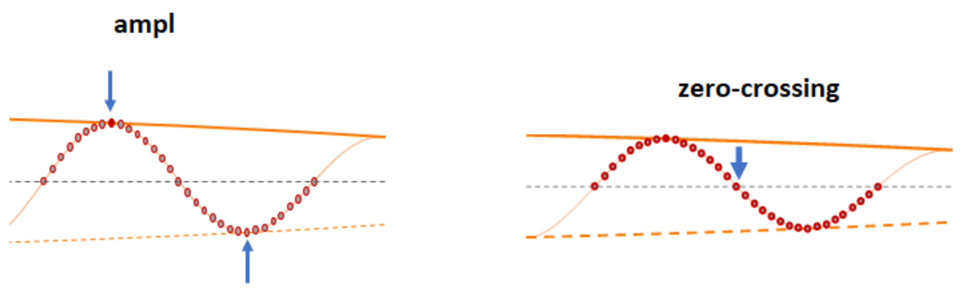


Figure 10. Input signal checker samples

The checks that are performed within input signal checker part are listed in the following table. Test #1 ensures the SDADC and eDMA are not stucked. Amplitude range check is watching the amplitude does not over-cross the given threshold. Mean value check tests if there is any unwanted DC shift present in the feedback signals. Zero crossing check watches if the signal in the buffer is phase-shifted. Plausibility check ensures the samples fits into the unitary circle.

Table 12. Input signal checks

1.	Check the sample values are changing.	cos[8](n)<>cos[8](n-1) cos[24](n)<>cos[24](n-1) sin[8](n)<>sin[8](n-1) sin[24](n)<>sin[24](n-1)

Table continues on the next page...

Table 12. Input signal checks (continued)

2.	Check amplitude range.	$\lvert\sin[8]\rvert < AMPL_{MAX}$ $\lvert\sin[24]\rvert < AMPL_{MAX}$ $\lvert\cos[8]\rvert < AMPL_{MAX}$ $\lvert\cos[24]\rvert < AMPL_{MAX}$
3.	Check signal mean value (DC shift).	$\lvert\sin[8] + \sin[24]\rvert < MEAN_{MAX}$ $\lvert\cos[8] + \cos[24]\rvert < MEAN_{MAX}$
4.	Check signal zero-crossing (phase shift)	$\lvert\sin[16]\rvert < ZC_{MAX}$ $\lvert\cos[16]\rvert < ZC_{MAX}$
5.	Check plausibility of samples (unit circle).	$\sin^2[8] + \cos^2[8] \cong const$ $\sin^2[24] + \cos^2[24] \cong const$

1.2.2 ATO checker

ATO checker uses the same samples as Input signal checker. It calculates the angle using an alternate method in order to avoid a systematic error. The angle is evaluated as arcus tangent of sine and cosine sample ratio for both half periods.

$$\theta \cong arctan\left(\frac{sin[8]}{cos[8]}\right) \qquad \theta \cong arctan\left(\frac{-sin[24]}{-cos[24]}\right)$$

The samples sin[24] and cos[24] are negated to get correct angle value. The arcus tangent function is implemented as simplified polynomial approximation. The maximum error is 0.22deg. All four quadrands are handled. The resulting angle is then compared with eTPU ATO result.

1.2.3 Extrapolation checker

Extrapolation checker reads and stores ATO outputs, angle updates per half period plus one extrapolated value. Checker use these samples and with a small delay checks whether the extrapolated angle fits between preceding and following ATO angle outputs. The acceptance range is extended for the case of small or zero speed when the preceding and the following ATO angles are almost the same.

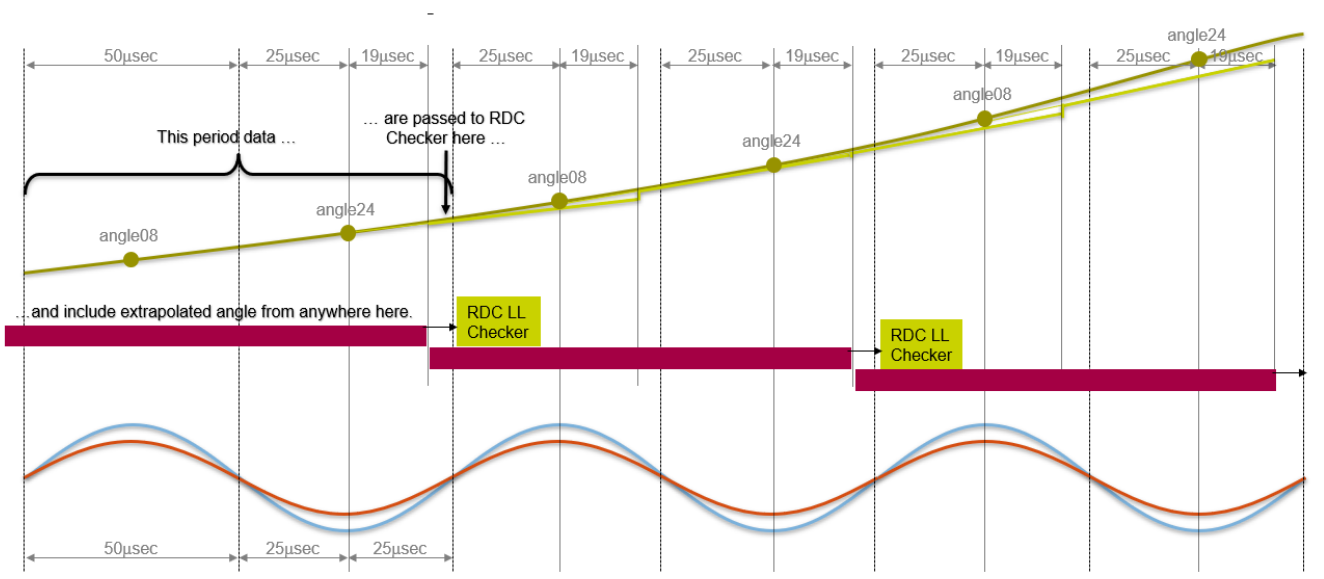


Figure 11. Extrapolation checker timing

1.2.4 Timing checker

The timing checker has two tasks, to check whether the timestapms corresponding to ATO angle updates are changing. This ensures that the ATO activity is not stucked.

timestamp[8](n) <> timestamp[8](n - 1)

timestamp[24](n) <> timestamp[24](n - 1)

The second task is to check whether the ATO updates are coming within expected time and thus samples are plausible.

$$timestamp[24](n) - timestamp[8](n) \sim \frac{PERIOD}{2}$$

$$timestamp[8](n) - timestamp[24](n-1) \sim \frac{PERIOD}{2}$$

Chapter 2
Integrating and Building the Product

The development environment includes:

- MPC5777C EVB
- S32 Design Studio (S32DS)
- Optionaly Safety Runtime Framework (SRF)

To successfully run the RDC Checker the following steps needs to be performed:

1. Run eTPU Resolver function. The recommended channel usage is to run ATO channel on different engine than Extrapolation channel.
2. Enable RDC Checker functionality by configuring following option in resolver_config_t structure, FS_ETPU_RESOLVER_OPTIONS_RDC_CHECKER_ON
3. Include the RDC Checker source (*rdc_checker.c/.h*) code into your project
4. Configure DMA to transfer data from eTPU RAM (located on *checker_signal_pba*) to *Rdc_InputData* structure used by RDC Checker.
5. Configure the thresholds of RDC Checker in *rdc_checker.h*
6. Enable DMA request from eTPU ATO channel and run *RDC_Checker_LL()* on IRQ from eDMA channel.
7. Use either *RDC_Checker_FaultStatus()* to read the RDC Checker results, or use the Safety Runtime Framwork (SRF) to integrate RDC Checker into a safety environment of application checkers and reactions.

The above mentioned steps are described in detail later in the sections.

2.1 How to get eTPU Resolver function up and running within CPU application

User has two options to get the eTPU Resolver binary, either download the compiled binary from NXP website (http://www.nxp.com/etpu) or use eTPU source code and eTPU compiler. AN5374 describes in detail how to integrate the eTPU binary into the CPU application.

To configure the resolver function you have to modify etpu_gct.c and etpu_gct.h files.

2.1.1 Configuration of etpu_gct.h

- Channel assignment is performed as per the following steps. Resolver excitation and ATO function is running on engine A, Extrapolation feature (SAMPLE_CHAN) is running on engine B. Diagnostics were set to engine B as well, but they are not used within this project.

```
/*============================================================================================
DEFINE FUNCTIONS TO CHANNELS
============================================================================================*/
#define ETPU_RESOLVER_EXC_CHAN ETPU_ENGINE_A_CHANNEL(0)
#define ETPU_RESOLVER_ATO_CHAN ETPU_ENGINE_A_CHANNEL(1)
#define ETPU_RESOLVER_DIAG_CHAN ETPU_ENGINE_B_CHANNEL(0)
#define ETPU_RESOLVER_SAMPLE_CHAN ETPU_ENGINE_B_CHANNEL(27)
```

- Configure both IRQ and DMA so both can be generated from ATO channel. DMA request is to transfer data from structure in eTPU data RAM to system RAM. IRQ is not essential for the RDC Checker functionality, it is used here within this example application to read calculated outputs and states from resolver function as well as update data from config structure to the eTPU Resolver.

```
/
*=================================================================================================*
DEFINE INTERRUPT ENABLE, DMA ENABLE AND OUTPUT DISABLE
===================================================================================================
*/
#define ETPU_CIE_A (1<<ETPU_RESOLVER_ATO_CHAN) /* enable interrupt on ETPU_RESOLVER_ATO_CHAN */
#define ETPU_DTRE_A (1<<ETPU_RESOLVER_ATO_CHAN) /* enable DMA request on ETPU_RESOLVER_ATO_CHAN */
#define ETPU_ODIS_A 0x00000000
#define ETPU_OPOL_A 0x00000000
#define ETPU_CIE_B 0x00000000
#define ETPU_DTRE_B 0x00000000
#define ETPU_ODIS_B 0x00000000
#define ETPU_OPOL_B 0x00000000
```

- Enable global variable access to make the eTPU structures being visible outside the gct files to be accessible from main function.

```
/
*=================================================================================================*
GLOBAL VARIABLE ACCESS
===================================================================================================
*/
/* Global RESOLVER structures defined in etpu_gct.c */
extern struct resolver_instance_t resolver_instance;
extern struct resolver_config_t resolver_config;
extern struct resolver_outputs_t resolver_outputs_calculated;
extern struct resolver_outputs_t resolver_outputs_extrapolated;
extern struct resolver_outputs_t resolver_outputs_mechanical;
extern struct resolver_diag_measures_t resolver_diag_measures;
extern struct resolver_diag_thresholds_t resolver_diag_thresholds;
extern struct resolver_diag_flags_t resolver_diag_flags;
extern struct resolver_states_t resolver_states;
```

- Declare function prototypes.

```
/
*==================================================================================================*

FUNCTION PROTOTYPES
===================================================================================================
*/
int32_t my_system_etpu_init (void);
void my_system_etpu_start(void);
```

2.1.2 Resolver function configuration etpu_gct.c

The Resolver function configuration itself is performed within *etpu_gct.c* together with general engine configuration which is already part of etpu_gct.c/h template files. This example configuration can be used as it is or you can modify as per your needs.

- Apart from the files that are already present in template files include device specific file to specify eTPU data RAM frame plus Resolver API header file.

```
/*================================================================================================*
INCLUDE FILES
=================================================================================================*/
#include "etpu_gct.h" /* private header file */
```

```
#include "etpu\_utils\etpu_util.h" /* General C Functions for the eTPU */
#include "etpu\_etpu_set\etpu_set.h" /* eTPU function set code binary image */
#include "etpu\resolver\etpu_resolver.h" /* eTPU Resolver function API */
#include "include/mpc5777c_vars.h" /* Device specific header file */
```

- eTPU engine configuration: Modify the predefined structure including parameters to configure eTPU engine. For a detailed of the eTPU engine configuration please refer to eTPU Reference manual.

```
/*=============================================================================================*
Global eTPU settings - etpu_config structure
=============================================================================================*/
/** @brief Structure handling configuration of all global settings */
struct etpu_config_t my_etpu_config =
       {
            …
       };
```

- Define Resolver instance structure: Uses definitions of particular channels from *etpu_gct.h* for particular functionality assignment. Configure the Resolver function priority. Start offset parameter determines the first rising edge time of the Excitation signal – it is scheduled *start_offset* number of TCR1 cycles after Resolver initialization. ADC delay is the parameter used to adjust the timestamp of ATO update. Start offset and ADC delay will be described in more detail in chapter Configurations. Channel parameter base address *cpba* can be configured for a particular address within the eTPU data memory (like in this case) or can be configured as 0, in that case *cpba* will be allocated automatically. The similar case applies for *signals_pba* which stands for feedback signals parameter base address, a place within eTPU data RAM where sampled feedback signals are transferred from Sigma-delta ADCs FIFO using DMA requests. Parameter *checker_signals_pba* is address of structure where particular signals are stored for the purpose of RDC Checker. This should be configured as 0 and if RDC Checker option is enabled the address is automatically allocated on initialization.

NOTE

The *resolver_instance_t* structure can be modified only once at initialization. It is not a subject of change during the function runtime.

```
/*============================================================================================*
eTPU channel settings - RESOLVER
============================================================================================*/
/** @brief Initialization of RESOLVER structures */
struct resolver_instance_t resolver_instance =
{
 ETPU_RESOLVER_EXC_CHAN, /* chan_num_exc */
 ETPU_RESOLVER_ATO_CHAN, /* chan_num_ato */
 ETPU_RESOLVER_DIAG_CHAN, /* chan_num_diag */
 ETPU_RESOLVER_SAMPLE_CHAN,/* chan_num_sample */
 ETPU_RESOLVER_ATO_CHAN, /* chan_num_dma - etpuA1 generate dma request on channel DMA_A 28 */
 FS_ETPU_PRIORITY_HIGH, /* priority */USEC2TCR1(100), /*
 start_offset */USEC2TCR1(19), /* adc_delay */
 0xC3FC8300, /* *cpba */ /* 0 for automatic allocation */
 0xC3FC8600, /* *signals_pba */ /* 0 for automatic allocation */
 0 /* *checker_signals_pba - will be allocated automatically */
};
```

- Define Resolver configuration structure: Use parameter *options* to configure intended functionality. *Excitaiton_period* is configured to 99,84 μs – this is configured with respect to SDADC output data rate. Refer to chapter Configurations for more details. Parameters *ato_p_gain* and *ato_i_gain* are constants of ATO PI controller, refer to *Resolver ATO PI-gains.xlsx* for the particular configurations. Excitation signal phase shift adjustment is controlled by PI controller with parameters *exc_p_gain* and *exc_i_gain* that are configured empirically. The configuration used in this example is recommended. Last parameter *q_ewma_speed* is an EWMA filter coefficient used to filter ATO speed updates. Is should be ~0.9 meaning the 0.9 weight is applied on previous filtered speed value.

```
struct resolver_config_t resolver_config =
{
 FS_ETPU_RESOLVER_SEMAPHORE_0, /* smpr_id */
 FS_ETPU_RESOLVER_OPTIONS_CALCULATION_ON +
 FS_ETPU_RESOLVER_OPTIONS_DIAG_MEASURES_ON +
 FS_ETPU_RESOLVER_OPTIONS_DIAG_FLAGS_ON +
 FS_ETPU_RESOLVER_OPTIONS_EXC_ADAPTATION_ON +
 FS_ETPU_RESOLVER_OPTIONS_EXC_GENERATION_ON+
 FS_ETPU_RESOLVER_OPTIONS_RDC_CHECKER_ON, /* options */
 NSEC2TCR1(99840), /* excitation_period */
 SFRACT24(0.070597541), /* ato_p_gain */
 SFRACT24(0.002492006), /* ato_i_gain */
 SFRACT24(0.00000), /* exc_p_gain */
 SFRACT24(0.00012), /* exc_i_gain */
 SFRACT24(0.99) /* q_ewma_speed */
};
```

- Define Resolver Diag thresholds structure: This is used when diagnostics running on eTPU are enabled. This structure contains thresholds that are compared against the diagnostic measures and diagnostic flags are set upon that comparison. For detailed description of each threshold and diagnostic measures please refer to chapter eTPU diagnostics.

```
struct resolver_diag_thresholds_t resolver_diag_thresholds =
{
 SFRACT24(0.6), /* ampl_thrs */
 SFRACT24(0.15), /* low_ampl_thrs */
 SFRACT24(0.05), /* mean_thrs */
 SFRACT24(0.4*0.4), /* vec_min_thrs */
 SFRACT24(0.6*0.6), /* vec_max_thrs */
 SFRACT24(0.1), /* ampl_diff_thrs */
 SFRACT24(5.0/360), /* ato_angle_err_thrs */
 RPM2FRACT(10000, 99840) /* ato_speed_max_thrs */
};
```

- Define rest of the structures containing ATO and diagnostics and states.

```
struct resolver_outputs_t         resolver_outputs_calculated;
struct resolver_outputs_t         resolver_outputs_extrapolated;
struct resolver_outputs_t         resolver_outputs_mechanical;
struct resolver_diag_measures_t   resolver_diag_measures;
struct resolver_diag_flags_t      resolver_diag_flags;
struct resolver_states_t          resolver_states;
```

- Implement the eTPU initialization – modify my_system_etpu_init() function with adding Resolver initialization from Resolver API. This function call comes after global eTPU initialization as follows.

```
/* Initialization of eTPU channel settings */
err_code = fs_etpu_resolver_init(&resolver_instance, &resolver_config);
if(err_code != FS_ETPU_ERROR_NONE) return(err_code + (ETPU_RESOLVER_EXC_CHAN<<16));
```

- Call the my_system_etpu_init() and my_system_etpu_start() in main function to get the configuration done and start eTPU. Note that also for proper Resolver operation it requires to configure and start DMA and SDADC as well. See the correct sequence of calling the routines below.

```
/* initialize eTPU */
my_system_etpu_init();
```

```
/* Initialize eDMA */
edma_init();
/* SDADC init*/
sdadc_init();
/* run the eTPU */
my_system_etpu_start();
/* start the DMA */
edma_start();
/* start the SDADC */
sdadc_start();
```

- Use the functions from Resolver API (etpu_resolver.c/.h) to interface Resolver function in runtime.

2.2 How to include RDC Checker into application

To correctly run the RDC Checker user has to get the Resolver function up and running (as described in previous chapter How to get eTPU Resolver function up and running within CPU application) at first. Then you have to include rdc_checker.c/.h files into the project and perform threshold configuration. Also DMA has to be configured to transfer data upon ATO Resolver eTPU channel request. All the necessary steps are further described in detail.

- Include RDC Checker files. The files should be included into the main.c.
- Configure DMA: The configuration of particular DMA channel that is triggered by eTPU ATO channel has to be performed to transfer data from eTPU data RAM into *Rdc_InputData* structure that is defined as a global variable within the *rdc_checker.c* file. For more details please refer to chapter DMA configuration for RDC Checker.
- Modify the RDC Checker thresholds that are defined in *rdc_checker.h* and can be modified by user according to application needs. The thresholds are equivalent to resolver diagnostic thresholds

```
/* RDC Checker fault thresholds */
#define RDC_THR_SIG_MAX_AMPL     RDC_FRACT16(0.8)
#define RDC_THR_SIG_DC_SHFT     RDC_FRACT16(0.05)
#define RDC_THR_SIG_ZC_AMPL     RDC_FRACT16(0.05)
#define RDC_THR_VEC_MIN_SQR     RDC_FRACT16(0.6 * 0.6)
#define RDC_THR_VEC_MAX_SQR     RDC_FRACT16(0.8 * 0.8)
#define RDC_THR_ATO_ERR         RDC_FRACT24(2.0/360.0)
#define RDC_THR_EXT             RDC_FRACT24(1.0/360.0)
#define RDC_THR_TIMING          (1997)
#define RDC_PERIOD_TCR1         (19968)
```

2.3 How to run RDC checker within application

There are two options how to use the RDC Checker within the application:

- Stand-alone checker: After *RDC_Checker_LL()* call *RDC_Checker_FaultStatus()* to get all detected fault flags and process them using accumulation, delay, decoupling and other techniques. The following figure shows the timing diagram.

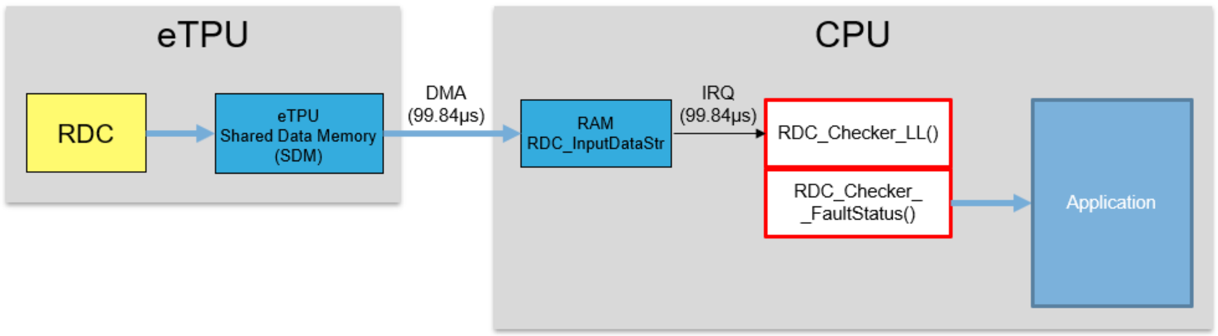


Figure 12. RDC checker standalone usage

The example can be found within the testing application *MPC5777C_RDC_Checker* attached to this document where the interrupt routine on DMA channel transfer completion is used to run the RDC Checker.

```
void DMA_Isr(void)
{
        …
        /* low level RDC Checker performs input data check and */
        RDC_Checker_LL();
        /* Read the fault status from LL RDC checker */
        RDC_fault = RDC_Checker_FaultStatus();

        if( RDC_fault == 0)
        {
                /* no fault - normal operation */
        }
        else
        {

                if(first_fault) /* First fault state detected - capture the transient fault */
                {
                        first_fault = RDC_fault;
                }
                else
                {
                        accumulated_faults |= RDC_fault;
                        fault_cnt++;

                        if(fault_cnt > 100)
                        {
                                /* Report to Safety manager */
                                RDC_Checker();
                                /* Clear counters and fault variables */
                                fault_cnt = 0;
                                first_fault = 0;
                                accumulated_faults = 0;
                        }
                }
        }
        /* Clear the interrupt flag */
        DMA_A.CINT.B.CINT = 28;
}
```

- RDC Checker used within the SRF: *RDC_Checker_LL()* runs on DMA interrupt request. The *RDC_Checker()* runs periodically as one of the 100µsec tasks and reports to SRF Safety Manager (see the following figure). The Safety Manager handles the reported faults, their decoupling and execution of safety reaction.

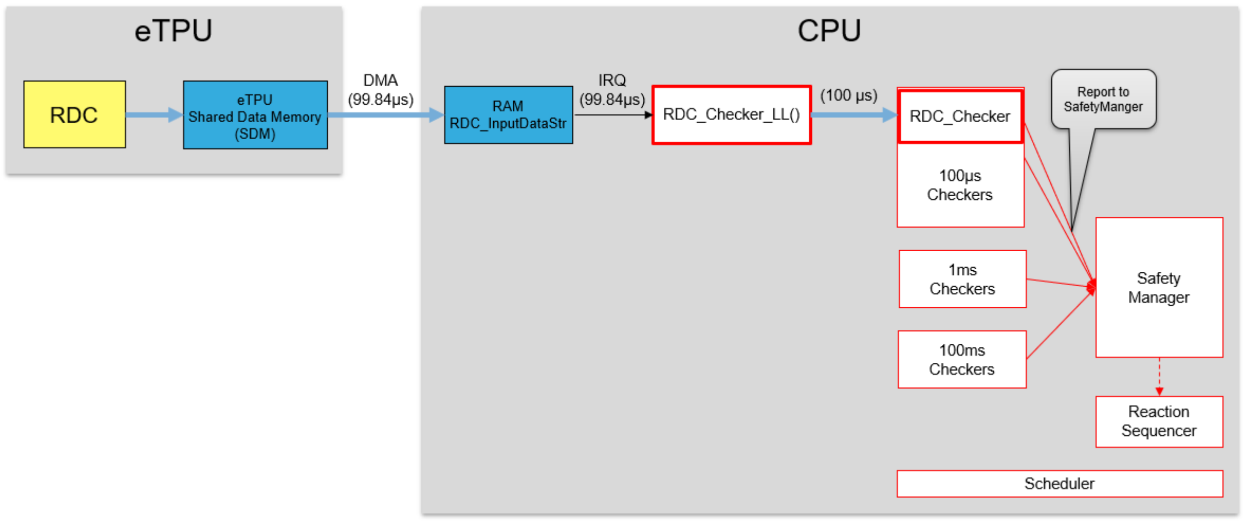


Figure 13. RDC checker usage within the Safety Runtime Framework

Chapter 3 Configurations

3.1 DMA configuration for RDC checker

Configuration of DMA channel that is triggered by Resolver ATO channel to transfer data from eTPU memory (resolver_instance. checker_signals_pba) to *Rdc_InputData* structure that is used by RDC Checker is described in the folloing table.

Table 13. DMA configuration for RDC checker

Configuration Item	RDC Checker DMA channel
Source address	resolver_instance.checker_signals_pba+ 0x4000 (reading from PSE memory region)
Destination address	&Rdc_InputData.sin08 (first member of *Rdc_InputData* structure)
Source transfer size / modulo	32-bits / 0 bytes
Destination transfer size / modulo	32-bits / 44 bytes
Source address offset	4 bytes
Destination address offset	4 bytes
Minor loop byte count	44 bytes
Major loop iteration count	1
Last source address adjustment	-44 bytes
Last destination address adjustment	-44 bytes
Channel to channel linking	disabled
Linked channel	-
Interrupt on major loop complete	enabled

3.2 SDADC config

Reffer to the description in chapter Sine and Cosine feedack signal digitization for the reference.

3.3 ADC delay parameter

The parameter *adc_delay* is evaluated based on SDADC group delay as well as ATO updated calculation time together with delay caused by transferring the samples from SDADC result FIFO into eTPU data memory using DMA. Another important information taken into account is that angle value calculated from sin/cos half-period N, corresponds to the time position of the next, N+1st, half-period center, because the ATO output is actually prediction of the angle for next half-period. The ADC delay is then evaluated as time difference between the ATO N end of update and the center of N+1 half-period. The evaluation is described in ATO timing and ADC delay parameter evaluation.

The SDADC group delay can be evaluated according to SDADC electrical specifications as follows.

$$Group\ delay\ [OSR = 24] = 235.5 * T_{clk}$$

$$T_{clk} = \frac{1}{(\frac{f_{ADCD_M}}{2})}$$

$$f_{ADCD_M} = \frac{200}{13}\ MHz = 15.38\ MHz \rightarrow T_{clk} = 130\ ns$$

$$Group\ delay\ [OSR = 24] = 235.5 * 130\ ns = 30.615\ \mu s$$

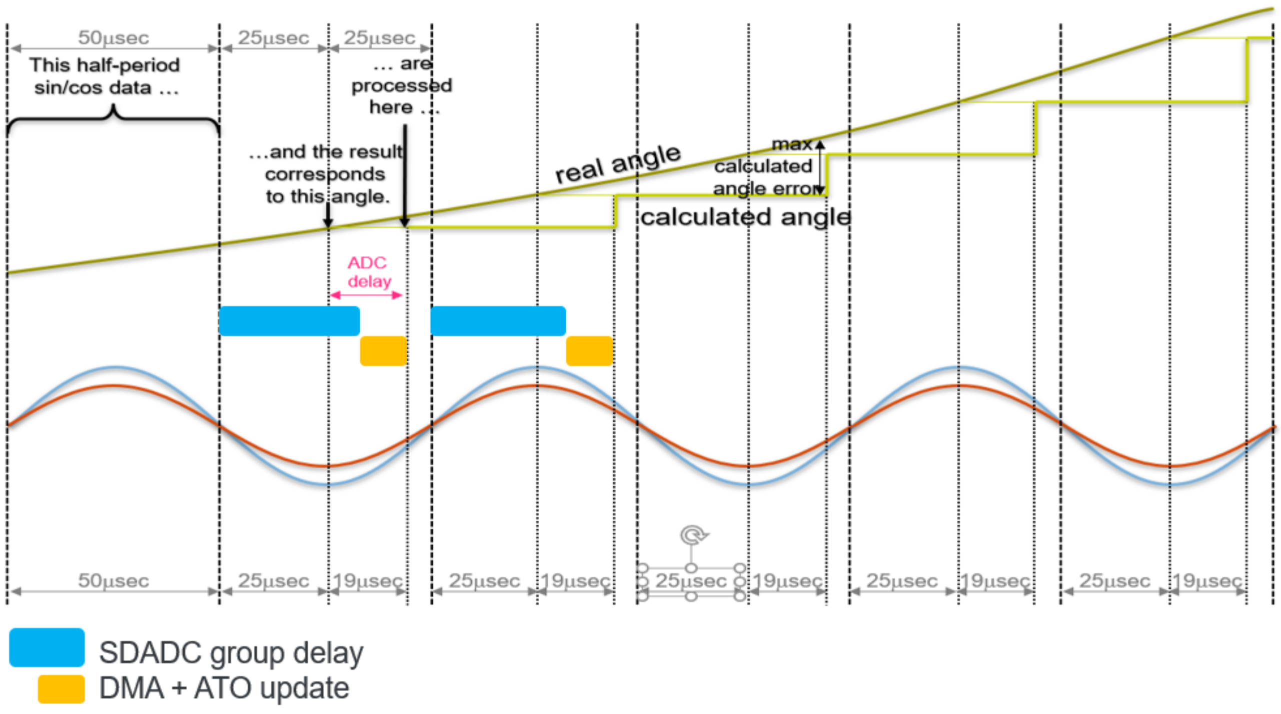


Figure 14. ATO timing and ADC delay parameter evaluation

3.4 Start offset configuration

Start_offset parameter determines the time when the generation of excitation signal starts after first ATO 1st host service request is received. It is important to evaluate this parameter since RDC function generates excitation signal as a PWM signal with 50 % duty cycle. This is then supposed to be fed into analog filters and further to resolver sensor excitation input. This HW chain causes a certain phase shift which is propagated into sine and cosine feedback signals. For the eTPU RDC it is crutial to have the feedback signals aligned in the buffers so the first-half period of the buffer contains the part of the signal corresponding to first half-period of excitation signal. At the eTPU RDC function startup the excitation signal phase shift is adjusted so the feedback signals are aligned in the signal buffers and zero-crossing error is minimized. The *start_offset* parameter has to be handled carefully to prevent locking the feedback signals into buffers with 180 degree error. The start_offset parameter and its relation to the excitation signal and feedback signals is illustrated in the following figure.

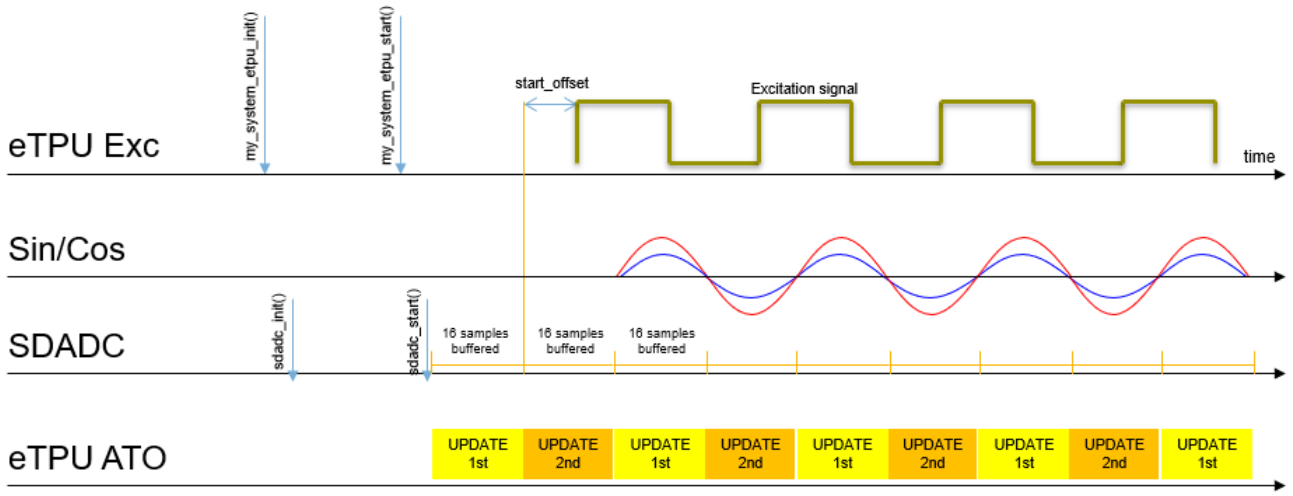


Figure 15. Resolver start_offset parameter setting

Chapter 4 APIs

4.1 eTPU Resolver API

The Resolver Digital Interface API is used to control and monitor the eTPU function using the following routines:

- fs_etpu_resolver_init()
- fs_etpu_resolver_config()
- fs_etpu_resolver_get_states()
- fs_etpu_resolver_get_outputs_calculated()
- fs_etpu_resolver_get_outputs_extrapolated()
- fs_etpu_resolver_trans_outputs_el_to_mech()
- fs_etpu_resolver_sample()
- fs_etpu_resolver_get_diag_measures()
- fs_etpu_resolver_get_diag_flags()
- fs_etpu_resolver_decode_diag_flags_basic()
- fs_etpu_resolver_decode_diag_flags_advanced()
- fs_etpu_resolver_diag_reset()
- fs_etpu_resolver_set_thresholds()
- fs_etpu_resolver_diag_calibrate()

These API functions handle the following data structures:

- resolver_instance
- resolver_config
- resolver_outputs
- resolver_diag_measures
- resolver_diag_thresholds
- resolver_diag_flags
- resolver_states

The data structures include the following fields:

NOTE

For a detailed API description refer to RESOLVER-DoxyDoc.chm or RESOLVER-DoxyDoc.zip which are included in AN5335SW.

Interaction among the API routines and data structures is shown in the following figure.

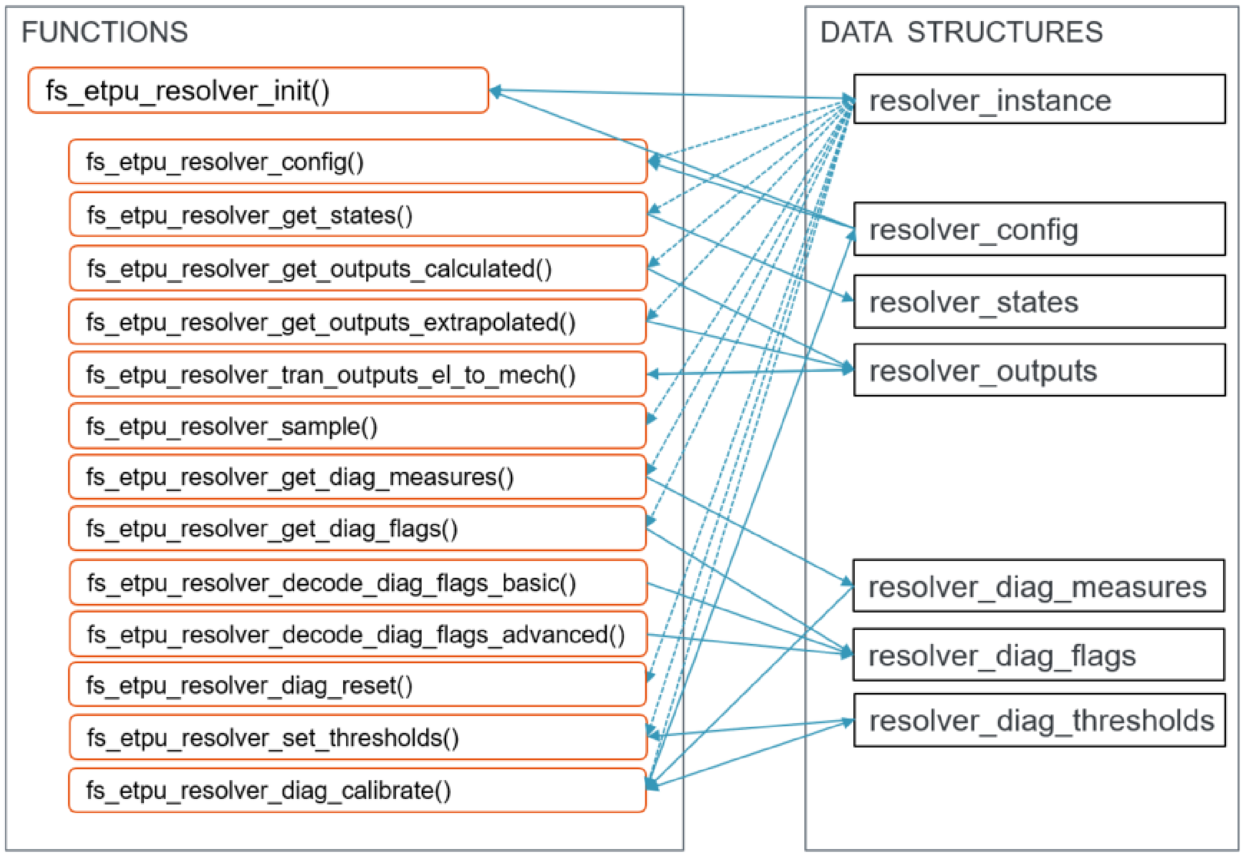

FUNCTIONS
fs_etpu_resolver_init()
fs_etpu_resolver_config()
fs_etpu_resolver_get_states()
fs_etpu_resolver_get_outputs_calculated()
fs_etpu_resolver_get_outputs_extrapolated()
fs_etpu_resolver_tran_outputs_el_to_mech()
fs_etpu_resolver_sample()
fs_etpu_resolver_get_diag_measures()
fs_etpu_resolver_get_diag_flags()
fs_etpu_resolver_decode_diag_flags_basic()
fs_etpu_resolver_decode_diag_flags_advanced()
fs_etpu_resolver_diag_reset()
fs_etpu_resolver_set_thresholds()
fs_etpu_resolver_diag_calibrate()
DATA STRUCTURES
resolver_instance
resolver_config
resolver_states
resolver_outputs
resolver_diag_measures
resolver_diag_flags
resolver_diag_thresholds

Figure 16. Resolver API functions and data structures

4.2 RDC Checker API

The RDC Checker API includes the following routines:

```
void Rdc_Checker_LL( void )
uint32_t Rdc_Checker_FaultStatus( void )
void Rdc_Checker( void )
uint32_t Rdc_FaultInj( void )
```

- *Rdc_Checker_LL* is the low-level checker routine which, on DMA IRQ, checks the eTPU data and set fault flags
- *Rdc_Checker_FaultStatus* returns the detected fault flags and cleans them. The fault flags are cumulated until cleaned by *Rdc_Checker_FaultStatus* or *Rdc_Checker*.
- *Rdc_Checker* is intended to report cumulated flags to SRF Safety Manager and clean them.
- *Rdc_FaultInj* is intended to be used within the SRF as part of a self-check by fault injection.

The RDC Checker detects the following fault flags (fault status).

```
/* RDC Checker fault flags */
#define RDC_FAULT_SIN_STUCK     (0x0001UL)
#define RDC_FAULT_COS_STUCK     (0x0002UL)
#define RDC_FAULT_SIN_AMPL_OOR  (0x0004UL)
#define RDC_FAULT_COS_AMPL_OOR  (0x0008UL)
#define RDC_FAULT_SIN_DC_SHFT   (0x0010UL)
#define RDC_FAULT_COS_DC_SHFT   (0x0020UL)
```

```
#define RDC_FAULT_SIG_PHS_SHFT  (0x0040UL)
#define RDC_FAULT_VEC_OOR       (0x0100UL)
#define RDC_FAULT_ATO_ERR       (0x0200UL)
#define RDC_FAULT_EXT_ERR       (0x0400UL)
#define RDC_FAULT_TIME_STUCK    (0x1000UL)
#define RDC_FAULT_TIMIMNG       (0x2000UL)
```

Chapter 5
Demos and Application Notes

5.1 RDC Checker testing application

The RDC Checker testing application *MPC5777C_RDC_Checker* is showing the use of Resolver function together with RDC Checker in stand-alone option. Note that in this application no real Resolver HW is used, the feedback signals are emulated within the PIT Timer interrupt routine and placed into eTPU data RAM on signal_pba address. Also the DMA together with SDADCs are not used here, HSR request are written into the Resolver ATO channel at the end of PIT Timer ISR by using Resolver API function. Within this appolication several test-cases are defined to demonstrate the RDC Checker functionality.

Chapter 6
References

Table 14. References

S. No.	Description	Location
1.	SRF_PIM_MPC5775E_BETA_0.9.0 (January 2019) Includes SRF, RDC Checker, and more.	www.nxp.com

Date of release: March 2020

Document identifier: eTPURDCCUG