ФРЕЙМВОРК CATFISH

Фрагмент исходного текста программы

СБЦТ.75408316.CatFish.12

1 ОБОЗНАЧЕНИЕ И НАИМЕНОВАНИЕ ПРОГРАММЫ

1.1. НАИМЕНОВАНИЕ

Полное наименование – фреймворк контроллера наведения 3-х осевой спутниковой антенной системы.

1.2. УСЛОВНОЕ ОБОЗНАЧЕНИЕ

Условное наименование – CatFish.

2 ФРАГМЕНТ ИСХОДНОГО ТЕКСТА ПРОГРАММЫ

Ниже приведен фрагмент исходного текста программы фреймворк CatFish.

```
package model
type AntennaStatus int
const (
	// AntennaStatusOff антенна отключена и не работает
	AntennaStatusOff		AntennaStatus = 0
	AntennaStatusInitialization AntennaStatus = 1

	// AntennaStatusHandbrake Антенна остановлена в ручном режиме
	AntennaStatusHandbrake AntennaStatus = 2
	// AntennaStatusStop Все приводы антенны остановлены
	AntennaStatusStop AntennaStatus = 3
	// AntennaStatusMove Как минимум 1 привод антенны в движении
	AntennaStatusMove AntennaStatus = 4
	// AntennaStatusGyrostabilizer Антенна в режиме гиростабилизации без направления на
спутник
	AntennaStatusGyrostabilizer AntennaStatus = 5
	// AntennaStatusGyrostabilizerDirectSatellite Антенна направлена на спутник с
гиростабилизацией (учитываются координаты платформы)
	AntennaStatusGyrostabilizerDirectSatellite AntennaStatus = 6

	AntennaStatusError	AntennaStatus = -1
	AntennaStatusFatalError AntennaStatus = math.MinInt
)

var instanceAntenna Antenna
var antennaOnce sync.Once
```

```
var NewAntenna = func() Antenna {
	antennaOnce.Do(func() {
		instanceAntenna = &antenna{
			status:              AntennaStatusOff,
			changeStatus:        make(chan AntennaStatus),
			algorithm:           NewIdleAlgorithm(),
			cancelAlgorithm:     func() {},
			changeAlgorithm:     make(chan Algorithm),
			antennaPosition:     newAntennaPosition(),
			satellite:           emptySatellite(),
			guidanceSignalReceiver:
guidance_signal_receiver.NewGuidanceSignalReceiver(),
			dvbReceiver:         dvb_receiver.NewDVBReceiver(),
			azimuth:             drivers.NewDriver(drivers.RoleAzimuth),
			seatAngle:           drivers.NewDriver(drivers.RoleSeatAngle),
			polarizationRx:      drivers.NewDriver(drivers.RolePolarizationRx),
			polarizationTx:      drivers.NewDriver(drivers.RolePolarizationTx),
			inertialNavigationSystem:
inertial_navigation_system.NewInertialNavigationSystem(),
		}
	})
	return instanceAntenna
}

type Antenna interface {
	// StatusCode возвращает текущий режим работы Антенны
	StatusCode() AntennaStatus
	// Algorithm Возвращает текущий алгоритм работы антенны
	Algorithm() Algorithm
	// Position Возвращает текущее положение антенны
	Position() AntennaPosition
	// Satellite Возвращает объект текущего спутника
	Satellite() Satellite
	// GuidanceSignalReceiver Возвращает текущий объект приемника сигнала наведения
	GuidanceSignalReceiver() guidance_signal_receiver.GuidanceSignalReceiver
	// DVBReceiver Возвращает текущий объект DVB приемника
	DVBReceiver() dvb_receiver.DVBReceiver
	// Azimuth Возвращает объект драйвера по азимуту
	Azimuth() drivers.Driver
	// SeatAngle Возвращает объект драйвера по углу места
	SeatAngle() drivers.Driver
	// PolarizationRx Возвращает объект драйвера по приемному поляризатору
	PolarizationRx() drivers.Driver
	// PolarizationTx Возвращает объект драйвера по передающему поляризатору
	PolarizationTx() drivers.Driver
	// InertialNavigationSystem Возвращает объект навигационной системы
```

```
	InertialNavigationSystem() inertial_navigation_system.InertialNavigationSystem

	// CheckAlgorithmRequirement проверяет возможность выполнения алгоритма согласно требованиям к железу
	CheckAlgorithmRequirement(alg Algorithm) error
	// RunAlgorithm Запускает алгоритм в работу. Если антенна занята другим алгоритмом, то возвращает ошибку
	RunAlgorithm(alg Algorithm) error
	// CancelAlgorithm Останавливает работу алгоритма
	CancelAlgorithm()

	// CreateChanPosition Подписка на текущую позицию антенны
	CreateChanPosition(size int) <-chan AntennaPosition
	// CloseChanPosition Отписка на текущую позицию антенны
	CloseChanPosition(ch <-chan AntennaPosition)
	// CreateChanStatus Подписка на текущий статус антенны
	CreateChanStatus(size int) <-chan AntennaStatus
	// CloseChanStatus Отписка от текущего статуса антенны
	CloseChanStatus(ch <-chan AntennaStatus)
	// CreateChanAlgorithm Подписка на смену алгоритмов
	CreateChanAlgorithm(size int) <-chan Algorithm
	// CloseChanAlgorithm Отписка от обновлений алгоритма
	CloseChanAlgorithm(ch <-chan Algorithm)

	// SetPosition устанавливает новую позицию антенны относительно нуля
	SetPosition(ctx context.Context, q common.Quaternion) error
	// SetHeadPosition устанавливает новую позицию антенны относительно носа платформы
	SetHeadPosition(ctx context.Context, q common.Quaternion) error
	// SetTruePosition устанавливает новую позицию антенны относительно истинного севера
	SetTruePosition(ctx context.Context, q common.Quaternion) error
	// SetHandbrake остановка всех драйверов антенны
	SetHandbrake(ctx context.Context, enable bool) error
	// SetStop Отключение режима гиростабилизации
	SetStop()
	// SetGyrostabilizer Включает режим гиростабилизации
	SetGyrostabilizer()
	// SetGyrostabilizerDirectSatellite Включает режим гиротабилизации с учетом перемещения платформы. Работает если настроен спутник
	SetGyrostabilizerDirectSatellite()
	// SetSatellite Устанавливает параметры текущего спутника
	SetSatellite(ctx context.Context, sat Satellite) error

	// Run Запускает антенну в работу
	Run(ctx context.Context) error
}

type antenna struct {
```

```
	// Общий статус антенны
	status AntennaStatus
	// Смена статуса антенны
	changeStatus chan AntennaStatus

	// Текущий алгоритм управления антенной
	algorithm Algorithm
	// Переключение алгоритма
	changeAlgorithm chan Algorithm
	// Принудительное завершение работы алгоритма
	cancelAlgorithm context.CancelFunc

	// Положение антенны в пространстве
	antennaPosition *antennaPosition
	// Параметры спутника
	satellite *satellite

	// Подключаемые устройства
	// Приемник сигнала наведения
	guidanceSignalReceiver guidance_signal_receiver.GuidanceSignalReceiver
	// DVB Receiver для определения истинного азимута с помощью телевизионного
приемника
	dvbReceiver dvb_receiver.DVBReceiver
	// Драйвер движения по азимуту
	azimuth drivers.Driver
	// Драйвер движения по углу места
	seatAngle drivers.Driver
	// Драйвер принимающего поляризатора
	polarizationRx drivers.Driver
	// Драйвер передающего поляризатора
	polarizationTx drivers.Driver
	// Навигационная система
	inertialNavigationSystem inertial_navigation_system.InertialNavigationSystem

	// Каналы для уведомления о текущей позиции антенны и платформы
	notifyAntennaPosition   []chan AntennaPosition
	muNotifyAntennaPosition sync.RWMutex
	// Канал для уведомления о текущем статусе контроллера
	notifyStatus   []chan AntennaStatus
	muNotifyStatus sync.RWMutex
	// Канал для уведомления о смене алгоритма
	notifyAlgorithm   []chan Algorithm
	muNotifyAlgorithm sync.RWMutex

	mu sync.RWMutex
}
```

```
// StatusCode возвращает статус Антенны
func (a *antenna) StatusCode() AntennaStatus {
	a.mu.RLock()
	defer a.mu.RUnlock()
	return a.status
}

// Algorithm Возвращает текущий активный алгоритм
func (a *antenna) Algorithm() Algorithm {
	a.mu.RLock()
	defer a.mu.RUnlock()
	return a.algorithm
}

// Position возвращает текущую позицию антенны
func (a *antenna) Position() AntennaPosition {
	return a.antennaPosition
}

// Satellite Возвращает объект текущего спутника
func (a *antenna) Satellite() Satellite {
	return a.satellite
}

// GuidanceSignalReceiver Возвращает текущий приемник системы наведения
func (a *antenna) GuidanceSignalReceiver() guidance_signal_receiver.GuidanceSignalReceiver {
	return a.guidanceSignalReceiver
}

// DVBReceiver Возвращает текущий приемник системы наведения
func (a *antenna) DVBReceiver() dvb_receiver.DVBReceiver {
	return a.dvbReceiver
}

// Azimuth Возвращает драйвер по азимуту
func (a *antenna) Azimuth() drivers.Driver {
	return a.azimuth
}

// SeatAngle Возвращает драйвер по углу места
func (a *antenna) SeatAngle() drivers.Driver {
	return a.seatAngle
}

// PolarizationRx Возвращает драйвер по поляризатору RX
func (a *antenna) PolarizationRx() drivers.Driver {
	return a.polarizationRx
```

```
}

// PolarizationTx Возвращает драйвер по поляризатору TX
func (a *antenna) PolarizationTx() drivers.Driver {
	return a.polarizationTx
}

// InertialNavigationSystem - возвращает ИНС
func (a *antenna) InertialNavigationSystem() inertial_navigation_system.InertialNavigationSystem {
	return a.inertialNavigationSystem
}

// CheckAlgorithmRequirement проверяем возможен ли запуск алгоритма на этой антенне
func (a *antenna) CheckAlgorithmRequirement(alg Algorithm) error {
	for _, r := range alg.Requirements() {
		switch r {
		case RequirementInternalNavigationSystem:
			if a.inertialNavigationSystem.Model() == inertial_navigation_system.ModelOff
{
				return fmt.Errorf("inertial navigation system off")
			}
		case RequirementGuidanceSignalReceiver:
			if a.guidanceSignalReceiver.Model() == guidance_signal_receiver.ModelOff {
				return fmt.Errorf("inertial navigation system off")
			}
		case RequirementDVBReceiver:
			if a.dvbReceiver.Model() == dvb_receiver.ModelOff {
				return fmt.Errorf("DVB receiver off")
			}
		default:
			// требование соблюдено
		}
	}
	return nil
}

// RunAlgorithm Запускает новый алгоритм управленья антенной
func (a *antenna) RunAlgorithm(alg Algorithm) error {
	antennaStatus := a.StatusCode()
	switch {
	case antennaStatus == AntennaStatusOff:
		return fmt.Errorf("antenna off")
	case antennaStatus == AntennaStatusInitialization:
		return fmt.Errorf("antenna initialization")
	case antennaStatus == AntennaStatusFatalError:
		return fmt.Errorf("antenna fatal error")
	case alg.StatusCode() <= AlgorithmStatusInitialization:
```

```
		return fmt.Errorf("this algorithm is not initialized")
	case alg.StatusCode() == AlgorithmStatusCompleted:
		return fmt.Errorf("this algorithm is completed")
	}

	// Проверяем возможность выполнения алгоритма
	if err := a.CheckAlgorithmRequirement(alg); err != nil {
		return err
	}

	select {
	case a.changeAlgorithm <- alg:
		return nil
	default:
		return fmt.Errorf("another algorithm is currently active")
	}
}

// CancelAlgorithm Отменяет текущую работу алгоритма
func (a *antenna) CancelAlgorithm() {
	a.mu.RLock()
	defer a.mu.RUnlock()
	if a.cancelAlgorithm != nil {
		a.cancelAlgorithm()
	}
}

// CreateChanPosition возвращает канал с уведомлениями о текущей позиции
func (a *antenna) CreateChanPosition(size int) <-chan AntennaPosition {
	a.muNotifyAntennaPosition.Lock()
	defer a.muNotifyAntennaPosition.Unlock()
	// Добавить mutex
	ch := make(chan AntennaPosition, size)
	a.notifyAntennaPosition = append(a.notifyAntennaPosition, ch)
	return ch
}

// CloseChanPosition закрывает канал для освобождения ресурсов
func (a *antenna) CloseChanPosition(ch <-chan AntennaPosition) {
	a.muNotifyAntennaPosition.Lock()
	defer a.muNotifyAntennaPosition.Unlock()
	for i, c := range a.notifyAntennaPosition {
		if ch == c {
			a.notifyAntennaPosition = append(a.notifyAntennaPosition[:i],
a.notifyAntennaPosition[i+1:]...)
			// Закрываем канал
			close(c)
```

```
                return
            }
        }
}

// CreateChanStatus возвращает канал с уведомлениями о текущем статусе антенны
func (a *antenna) CreateChanStatus(size int) <-chan AntennaStatus {
        a.muNotifyStatus.Lock()
        defer a.muNotifyStatus.Unlock()
        ch := make(chan AntennaStatus, size)
        a.notifyStatus = append(a.notifyStatus, ch)
        return ch
}

// CloseChanStatus закрывает канал для освобождения ресурсов
func (a *antenna) CloseChanStatus(ch <-chan AntennaStatus) {
        a.muNotifyStatus.Lock()
        defer a.muNotifyStatus.Unlock()
        for i, c := range a.notifyStatus {
            if ch == c {
                a.notifyStatus = append(a.notifyStatus[:i], a.notifyStatus[i+1:]...)
                // Закрываем канал
                close(c)
                return
            }
        }
}

// CreateChanAlgorithm возвращает канал с уведомлениями о работе нового алгоритма
func (a *antenna) CreateChanAlgorithm(size int) <-chan Algorithm {
        a.muNotifyAlgorithm.Lock()
        defer a.muNotifyAlgorithm.Unlock()
        ch := make(chan Algorithm, size)
        a.notifyAlgorithm = append(a.notifyAlgorithm, ch)
        return ch
}

// CloseChanAlgorithm закрывает канал для освобождения ресурсов
func (a *antenna) CloseChanAlgorithm(ch <-chan Algorithm) {
        a.muNotifyAlgorithm.Lock()
        defer a.muNotifyAlgorithm.Unlock()
        for i, c := range a.notifyAlgorithm {
            if ch == c {
                a.notifyAlgorithm = append(a.notifyAlgorithm[:i], a.notifyAlgorithm[i+1:]...)
                // Закрываем канал
                close(c)
                return
```

```
                }
        }
}

// SetPosition устанавливает новую позицию антенны относительно нуля антенны
func (a *antenna) SetPosition(ctx context.Context, q common.Quaternion) error {
        qDrivers := a.antennaPosition.setPosition(q)
        return a.setDriversMotion(ctx, qDrivers, common.Vector3{})
}

// SetHeadPosition устанавливает новую позицию антенны относительно носа мобильного
объекта
func (a *antenna) SetHeadPosition(ctx context.Context, q common.Quaternion) error {
        qDrivers := a.antennaPosition.setHeadPosition(q)
        return a.setDriversMotion(ctx, qDrivers, common.Vector3{})
}

// SetTruePosition устанавливает новую позицию антенны относительно истинного севера
func (a *antenna) SetTruePosition(ctx context.Context, q common.Quaternion) error {
        qDrivers := a.antennaPosition.setTruePosition(q)
        return a.setDriversMotion(ctx, qDrivers, common.Vector3{})
}

// SetHandbrake Программная остановка драйверов
func (a *antenna) SetHandbrake(ctx context.Context, enable bool) error {
        // отправляем движки в стоп
        // TODO можно ускорить -> отправлять параллельно
        if err := a.azimuth.SetHandbrake(ctx, enable); err != nil {
                return err
        }
        if err := a.seatAngle.SetHandbrake(ctx, enable); err != nil {
                return err
        }
        if err := a.polarizationRx.SetHandbrake(ctx, enable); err != nil {
                return err
        }
        if err := a.polarizationTx.SetHandbrake(ctx, enable); err != nil {
                return err
        }
        return nil
}

// SetStop Выключает режимы гиростабилизации
func (a *antenna) SetStop() {
        a.changeStatus <- AntennaStatusStop
}
```

```
// SetGyrostabilizer Запускает режим гиростабилизации
func (a *antenna) SetGyrostabilizer() {
	a.changeStatus <- AntennaStatusGyrostabilizer
}

// SetGyrostabilizerDirectSatellite Запускает режим гиростабилизации + перерасчет положения спутника в зависимости от координат
func (a *antenna) SetGyrostabilizerDirectSatellite() {
	a.changeStatus <- AntennaStatusGyrostabilizerDirectSatellite
}

// SetSatellite устанавливает новые параметры спутников и отправляет в двигатели новые углы
func (a *antenna) SetSatellite(ctx context.Context, sat Satellite) error {

	// Настраиваем ПСН
	if err := a.guidanceSignalReceiver.SetFrequency(ctx, sat.Frequency()); err != nil {
		return err
	}

	// Настраиваем DVB ресивер
	if err := a.dvbReceiver.SetFrequency(ctx, sat.Frequency()); err != nil {
		return err
	}

	// Сохраняем новый спутник
	a.satellite.changeSatellite(sat)

	// Передаем двигателям новую цель
	if err := a.SetTruePosition(ctx,
a.Satellite().TruePosition(a.Position().Platform().Coordinates())); err != nil {
		return err
	}
	return nil
}

// Run Запускает процесс подключения к драйверам и отслеживания их
func (a *antenna) Run(ctx context.Context) error {
	if a.StatusCode() != AntennaStatusOff {
		return fmt.Errorf("antenna is work")
	}

	ctx, cancel := context.WithCancel(ctx)
	// Если функция завершает работу, то антенна переходит в статус инициализации
	a.setStatus(AntennaStatusInitialization)
	defer a.setStatus(AntennaStatusOff)

	chDone := make(chan error, 3)
```

```
	go func() {
		chDone <- a.listenStatus(ctx)
		cancel()
	}()
	go func() {
		chDone <- a.listenPosition(ctx)
		cancel()
	}()
	go func() {
		chDone <- a.executionAlgorithm(ctx)
		cancel()
	}()
	err := <-chDone
	_ = <-chDone
	_ = <-chDone

	// Закрываем каналы уведомлений и очищаем их
	a.muNotifyAlgorithm.Lock()
	for _, ch := range a.notifyAlgorithm {
		close(ch)
	}
	a.notifyAlgorithm = nil
	a.muNotifyAlgorithm.Unlock()

	a.muNotifyAntennaPosition.Lock()
	for _, ch := range a.notifyAntennaPosition {
		close(ch)
	}
	a.notifyAntennaPosition = nil
	a.muNotifyAntennaPosition.Unlock()

	a.muNotifyStatus.Lock()
	for _, ch := range a.notifyStatus {
		close(ch)
	}
	a.notifyStatus = nil
	a.muNotifyStatus.Unlock()

	return err
}

// executionAlgorithm загрузчик алгоритмов наведения
func (a *antenna) executionAlgorithm(ctx context.Context) error {
	var ctxAlg context.Context
	for {
		select {
		case alg := <-a.changeAlgorithm:
```

```
                        a.mu.Lock()
                        a.algorithm = alg
                        ctxAlg, a.cancelAlgorithm = context.WithCancel(ctx)
                        a.mu.Unlock()
                        // Уведомляем о смене алгоритма
                        a.sendNotifyChanAlgorithm(alg)
                        alg.Run(ctxAlg, a)
                        a.cancelAlgorithm()
                case <-ctx.Done():
                        a.CancelAlgorithm()
                        return ctx.Err()
                }
        }
}

func (a *antenna) listenStatus(ctx context.Context) error {
        if gsrSimulator, ok := a.guidanceSignalReceiver.(*guidance_signal_receiver.Simulator); ok {
                gsrSimulator.SetCallbackPosition(func() (trueSatellite common.Quaternion,
trueAntenna common.Quaternion) {
                        return a.satellite.TruePosition(a.Position().Platform().Coordinates()),
a.Position().TruePosition()
                })
                defer gsrSimulator.DeleteCallbackPosition()
        }

        if dvbReceiver, ok := a.dvbReceiver.(*dvb_receiver.Simulator); ok {
                dvbReceiver.SetCallbackPosition(func() (trueSatellite common.Quaternion, trueAntenna
common.Quaternion) {
                        return a.satellite.TruePosition(a.Position().Platform().Coordinates()),
a.Position().TruePosition()
                })
                defer dvbReceiver.DeleteCallbackPosition()
        }

        log := common.ContextLog(ctx)

        if err := a.azimuth.Connect(common.ContextWithLog(ctx, log.WithField(common.DeviceKey,
"driver azimuth"))); err != nil {
                return err
        }
        azmCh := a.azimuth.CreateChanStatus(1)
        defer a.azimuth.CloseChanStatus(azmCh)
        azmStatus := a.azimuth.StatusCode()

        if err := a.seatAngle.Connect(common.ContextWithLog(ctx,
log.WithField(common.DeviceKey, "driver seat angle"))); err != nil {
                return err
```

```
	}
	saCh := a.seatAngle.CreateChanStatus(1)
	defer a.seatAngle.CloseChanStatus(saCh)
	saStatus := a.seatAngle.StatusCode()

	if err := a.polarizationRx.Connect(common.ContextWithLog(ctx,
log.WithField(common.DeviceKey, "driver polRx"))); err != nil {
		return err
	}
	pRxCh := a.polarizationRx.CreateChanStatus(1)
	defer a.polarizationRx.CloseChanStatus(pRxCh)
	pRxStatus := a.polarizationRx.StatusCode()

	if err := a.polarizationTx.Connect(common.ContextWithLog(ctx,
log.WithField(common.DeviceKey, "driver polTx"))); err != nil {
		return err
	}
	pTxCh := a.polarizationTx.CreateChanStatus(1)
	defer a.polarizationTx.CloseChanStatus(pTxCh)
	pTxStatus := a.polarizationTx.StatusCode()

	if err := a.inertialNavigationSystem.Connect(common.ContextWithLog(ctx,
log.WithField(common.DeviceKey, "INS"))); err != nil {
		return err
	}
	insCh := a.inertialNavigationSystem.CreateChanStatus(1)
	defer a.inertialNavigationSystem.CloseChanStatus(insCh)
	insStatus := a.inertialNavigationSystem.StatusCode()

	// Приемник сигнала наведения
	if err := a.guidanceSignalReceiver.Connect(common.ContextWithLog(ctx,
log.WithField(common.DeviceKey, "GSR"))); err != nil {
		return err
	}
	gsrCh := a.guidanceSignalReceiver.CreateChanStatus(1)
	defer a.guidanceSignalReceiver.CloseChanStatus(gsrCh)
	gsrStatus := a.guidanceSignalReceiver.StatusCode()

	// Приемник сигнала наведения
	if err := a.dvbReceiver.Connect(common.ContextWithLog(ctx,
log.WithField(common.DeviceKey, "DVBr"))); err != nil {
		return err
	}
	dvbRCh := a.dvbReceiver.CreateChanStatus(1)
	defer a.dvbReceiver.CloseChanStatus(dvbRCh)
	dvbRStatus := a.dvbReceiver.StatusCode()
```

```
	// Кешируем статус устройства чтоб не блочить мютексом
	antennaStatus := a.StatusCode()
	var err error
	antennaStatus, err = a.checkStatus(antennaStatus, azmStatus, saStatus, pRxStatus, pTxStatus,
gsrStatus, insStatus, dvbRStatus)

	for {
		// Ждем изменения позиций драйверов и обновляем позицию антенны
		select {
		// Драйвера
		case status, ok := <-azmCh:
			if !ok {
				return fmt.Errorf("azimuth driver notifications are closed")
			}
			azmStatus = status
		case status, ok := <-saCh:
			if !ok {
				return fmt.Errorf("seat angle driver notifications are closed")
			}
			saStatus = status
		case status, ok := <-pRxCh:
			if !ok {
				return fmt.Errorf("polarization rx driver notifications are closed")
			}
			pRxStatus = status
		case status, ok := <-pTxCh:
			if !ok {
				return fmt.Errorf("polarization rx driver notifications are closed")
			}
			pTxStatus = status
		// Навигационная система
		case status, ok := <-insCh:
			if !ok {
				return fmt.Errorf("INS notifications are closed")
			}
			insStatus = status
		case status, ok := <-gsrCh:
			if !ok {
				return fmt.Errorf("guidance signal receiver notifications status are
closed")
			}
			gsrStatus = status
		case status, ok := <-dvbRCh:
			if !ok {
				return fmt.Errorf("dvb receiver notifications status are closed")
			}
			dvbRStatus = status
```

```
                case status := <-a.changeStatus:
                        // Если меняется статус антенны
                        antennaStatus = status
                // Завершение работы
                case <-ctx.Done():
                        return ctx.Err()
                }

                antennaStatus, err = a.checkStatus(antennaStatus, azmStatus, saStatus, pRxStatus,
pTxStatus, gsrStatus, insStatus, dvbRStatus)
                if err != nil {
                        return err
                }
        }
}

// Функция принимает решения относительно статуса антенны
func (a *antenna) checkStatus(antennaStatus AntennaStatus, azmStatus, saStatus, pRxStatus, pTxStatus
drivers.Status, gsrStatus guidance_signal_receiver.Status, insStatus inertial_navigation_system.Status,
dvbRStatus dvb_receiver.Status) (AntennaStatus, error) {
        // Принимаем решения насчет глобального статуса
        switch {
        case antennaStatus == AntennaStatusOff:
                return antennaStatus, fmt.Errorf("antenna off")
        // Не восстановимая ошибка завершаем работу
        case azmStatus == drivers.StatusFatalError:
                antennaStatus = a.setStatus(AntennaStatusFatalError)
                return antennaStatus, fmt.Errorf("azimuth driver fatal error")
        case saStatus == drivers.StatusFatalError:
                antennaStatus = a.setStatus(AntennaStatusFatalError)
                return antennaStatus, fmt.Errorf("seat angle driver fatal error")
        case pRxStatus == drivers.StatusFatalError:
                antennaStatus = a.setStatus(AntennaStatusFatalError)
                return antennaStatus, fmt.Errorf("pol rx driver fatal error")
        case pTxStatus == drivers.StatusFatalError:
                antennaStatus = a.setStatus(AntennaStatusFatalError)
                return antennaStatus, fmt.Errorf("pol tx driver fatal error")
        case insStatus == inertial_navigation_system.StatusFatalError:
                antennaStatus = a.setStatus(AntennaStatusFatalError)
                return antennaStatus, fmt.Errorf("inertial navigation system fatal error")
        case gsrStatus == guidance_signal_receiver.StatusFatalError:
                antennaStatus = a.setStatus(AntennaStatusFatalError)
                return antennaStatus, fmt.Errorf("guidance signal receiver fatal error")
        case dvbRStatus == dvb_receiver.StatusFatalError:
                antennaStatus = a.setStatus(AntennaStatusFatalError)
                return antennaStatus, fmt.Errorf("dvb reciever fatal error")
        case antennaStatus == AntennaStatusFatalError:
```

```
			antennaStatus = a.setStatus(AntennaStatusFatalError)
			return antennaStatus, fmt.Errorf("antenna fatal error")

		// Ошибку можно восстановить
		case azmStatus < drivers.StatusInitialization || saStatus < drivers.StatusInitialization || pRxStatus
< drivers.StatusInitialization || pTxStatus < drivers.StatusInitialization || insStatus <
inertial_navigation_system.StatusInitialization || gsrStatus <
guidance_signal_receiver.StatusInitialization || dvbRStatus < dvb_receiver.StatusInitialization:
			antennaStatus = a.setStatus(AntennaStatusError)

		// Антенна остановлена в ручном режиме
		case azmStatus == drivers.StatusHandbrake && saStatus == drivers.StatusHandbrake &&
pRxStatus == drivers.StatusHandbrake && pTxStatus == drivers.StatusHandbrake:
			antennaStatus = a.setStatus(AntennaStatusHandbrake)

		// Процесс инициализации антенны
		case azmStatus == drivers.StatusInitialization || saStatus == drivers.StatusInitialization ||
pRxStatus == drivers.StatusInitialization || pTxStatus == drivers.StatusInitialization || insStatus ==
inertial_navigation_system.StatusInitialization || gsrStatus ==
guidance_signal_receiver.StatusInitialization || dvbRStatus == dvb_receiver.StatusInitialization:
			antennaStatus = a.setStatus(AntennaStatusInitialization)

		case antennaStatus == AntennaStatusGyrostabilizer:
			antennaStatus = a.setStatus(AntennaStatusGyrostabilizer)

		case antennaStatus == AntennaStatusGyrostabilizerDirectSatellite:
			antennaStatus = a.setStatus(AntennaStatusGyrostabilizerDirectSatellite)

		// Определяем что антенна остановлена если все драйвера остановлены
		case azmStatus <= drivers.StatusStop && saStatus <= drivers.StatusStop && pRxStatus <=
drivers.StatusStop && pTxStatus <= drivers.StatusStop:
			antennaStatus = a.setStatus(AntennaStatusStop)

		// Определяем что антенна в движении если хоть один драйвер в процессе движения
		case azmStatus > drivers.StatusStop || saStatus > drivers.StatusStop || pRxStatus >
drivers.StatusStop || pTxStatus > drivers.StatusStop:
			antennaStatus = a.setStatus(AntennaStatusMove)

		default:
			return antennaStatus, fmt.Errorf("не удалось определить статус антенны")
		}

		return antennaStatus, nil
}

// listenPosition Слушает перемещения двигателей, при получении новых данных от INS
передаем компенсацию на двигатели
```

```
func (a *antenna) listenPosition(ctx context.Context) error {
	// Получаем каналы уведомлений драйверов
	azmCh := a.azimuth.CreateChanAngleRad(1)
	defer a.azimuth.CloseChanAngleRad(azmCh)
	saCh := a.seatAngle.CreateChanAngleRad(1)
	defer a.seatAngle.CloseChanAngleRad(saCh)
	pRxCh := a.polarizationRx.CreateChanAngleRad(1)
	defer a.polarizationRx.CloseChanAngleRad(pRxCh)
	pTxCh := a.polarizationTx.CreateChanAngleRad(1)
	defer a.polarizationTx.CloseChanAngleRad(pTxCh)

	// Канал уведомления ИНС
	insCh := a.inertialNavigationSystem.CreateChanPosition(1)
	defer a.inertialNavigationSystem.CloseChanPosition(insCh)

	// Нулевые углы которые будут обновляться и высчитывать финальный кватернион
	y := common.FromYAxisAngleRadian(a.azimuth.AngleRad())
	x := common.FromXAxisAngleRadian(a.seatAngle.AngleRad())
	z := common.FromZAxisAngleRadian(a.polarizationRx.AngleRad())

	for {
		// Ждем изменения позиций драйверов и обновляем позицию антенны
		select {
		// Драйвера
		case angle, ok := <-azmCh:
			if !ok {
				return fmt.Errorf("azimuth driver notifications are closed")
			}
			y = common.FromYAxisAngleRadian(angle)
		case angle, ok := <-saCh:
			if !ok {
				return fmt.Errorf("seat angle driver notifications are closed")
			}
			x = common.FromXAxisAngleRadian(angle)
		case angle, ok := <-pRxCh:
			if !ok {
				return fmt.Errorf("polarization rx driver notifications are closed")
			}
			z = common.FromZAxisAngleRadian(angle)
		case _, ok := <-pTxCh:
			if !ok {
				return fmt.Errorf("polarization rx driver notifications are closed")
			}
		// Навигационная система
		case p, ok := <-insCh:
			if !ok {
				return fmt.Errorf("INS notifications are closed")
```

```
                        }

                        switch a.StatusCode() {
                        case AntennaStatusGyrostabilizer:
                                // Компенсируем смещение платформы если находимся в состоянии
гиростабилизации
                                q := a.antennaPosition.gyrostabilizer(p)
                                // Векторные скорости сразу меняем на противоположные
                                if err := a.setDriversMotion(ctx, q, p.AngularVelocity.Rotate(q).Scale(-
1)); err != nil {
                                        logrus.Errorf("new driver position %s", err)
                                }
                        case AntennaStatusGyrostabilizerDirectSatellite:
                                // Компенсируем смещение платформы если находимся в состоянии
гиростабилизации
                                q :=
a.antennaPosition.gyrostabilizerNewTrueTarget(a.Satellite().TruePosition(p.Coordinates), p)
                                // Векторные скорости сразу меняем на противоположные
                                if err := a.setDriversMotion(ctx, q, p.AngularVelocity.Rotate(q).Scale(-
1)); err != nil {
                                        logrus.Errorf("new driver position %s", err)
                                }
                        default:
                                a.antennaPosition.updateTrueTarget(p)
                        }

                // Завершение работы
                case <-ctx.Done():
                        return ctx.Err()
                }

                // Обновляем позицию антенны
                a.antennaPosition.updateCurrentPosition(y.Multiplies(x, z))

                // Отправляем уведомления о новой позиции антенны
                a.sendNotifyChanPosition(a.antennaPosition.copy())
        }

}

// setDriversMotion отправляет в драйверы новые углы
func (a *antenna) setDriversMotion(ctx context.Context, q common.Quaternion, angVel
common.Vector3) error {
        // Если устройство не инициализировано или ошибка, то сразу возвращаем ошибку
        a.mu.RLock()
        defer a.mu.RUnlock()
```

```
	// Нельзя управлять двигателем во время инициализации и ошибки
	switch a.status {
	case AntennaStatusInitialization:
		return fmt.Errorf("error initialization")
	case AntennaStatusError:
		return fmt.Errorf("error")
	case AntennaStatusFatalError:
		return fmt.Errorf("fatal error")
	}

	// Отправляем новые углы в драйвера
	y, x, z := q.ToEulerAnglesRadian()
	if err := a.azimuth.SetAngleRad(ctx, y, angVel[1]); err != nil {
		return err
	}
	if err := a.seatAngle.SetAngleRad(ctx, x, angVel[0]); err != nil {
		return err
	}
	if err := a.polarizationRx.SetAngleRad(ctx, z, angVel[2]); err != nil {
		return err
	}
	if err := a.polarizationTx.SetAngleRad(ctx, z, angVel[2]); err != nil {
		return err
	}
	return nil
}

// setStatus устанавливает новый статус контроллера движений
func (a *antenna) setStatus(status AntennaStatus) AntennaStatus {
	a.mu.Lock()
	defer a.mu.Unlock()
	if a.status != status {
		a.status = status
		a.sendNotifyChanStatus(a.status)
	}
	return a.status
}

// sendNotifyChanPosition отправляет уведомления о позиции антенны
func (a *antenna) sendNotifyChanPosition(pos AntennaPosition) {
	a.muNotifyAntennaPosition.RLock()
	defer a.muNotifyAntennaPosition.RUnlock()
	// Уведомляем всех подписчиков о новой позиции антенны или навигационной системы
	for _, ch := range a.notifyAntennaPosition {
		select {
		case ch <- pos:
		default:
```

```
                }
        }
}

// sendNotifyChanPosition отправляет уведомления о позиции антенны
func (a *antenna) sendNotifyChanStatus(s AntennaStatus) {
        a.muNotifyStatus.RLock()
        defer a.muNotifyStatus.RUnlock()
        // Уведомляем всех подписчиков о новой позиции антенны или навигационной системы
        for _, ch := range a.notifyStatus {
                select {
                case ch <- s:
                default:
                }
        }
}

// sendNotifyChanPosition отправляет уведомления о позиции антенны
func (a *antenna) sendNotifyChanAlgorithm(alg Algorithm) {
        a.muNotifyAlgorithm.RLock()
        defer a.muNotifyAlgorithm.RUnlock()
        // Уведомляем всех подписчиков о завершении работы алгоритма
        for _, ch := range a.notifyAlgorithm {
                select {
                case ch <- alg:
                default:
                }
        }
}

package model

import (
        "github.com/schnack/catfish/app/common"
        "github.com/schnack/catfish/app/model/inertial_navigation_system"
        "github.com/schnack/catfish/config"
        "math"
        "sync"
)

const DeltaAngle float64 = 0.000000004

func newAntennaPosition() *antennaPosition {
        return &antennaPosition{
                platform:   newPlatform(),
                zero:       common.IdentityQuaternion,
                head:
```

```
common.FromEulerAnglesDegrees(config.New().Antenna.Drivers.Offset.Course,
config.New().Antenna.Drivers.Offset.Pitch, config.New().Antenna.Drivers.Offset.Roll),
                trueTarget: common.IdentityQuaternion,
                relative:   config.New().Antenna.Drivers.Relative,
        }
}

// FactoryAntennaPosition Создает объект позиции антенны
func FactoryAntennaPosition(p Platform, zero, head, trueTarget common.Quaternion, relative bool)
AntennaPosition {
        pl, ok := p.(*platform)
        if !ok {
                pl = newPlatform()
        }
        return &antennaPosition{
                platform:   pl,
                trueTarget: trueTarget,
                head:       head,
                zero:       zero,
                relative:   relative,
        }
}

type AntennaPosition interface {
        Position() common.Quaternion
        HeadPosition() common.Quaternion
        TruePosition() common.Quaternion
        TargetPosition() common.Quaternion
        HeadTargetPosition() common.Quaternion
        TrueTargetPosition() common.Quaternion
        TrueMisalignment() float64
        HeadMisalignment() float64
        Misalignment() float64
        Platform() Platform
}

type antennaPosition struct {
        // Платформа
        platform *platform
        // Положение к которому стремиться антенна относительно севера
        trueTarget common.Quaternion
        // Положение нуля антенны относительно носа платформы
        head common.Quaternion
        // Положение антенны относительно нулевого положения антенны (текущее положение
антенны)
        zero common.Quaternion
```

```
	// Заменяет режим указания углов драйверам с абсолютных на относительные
	relative bool

	// Синхронизация
	mu sync.RWMutex
}

func (p *antennaPosition) copy() *antennaPosition {
	p.mu.RLock()
	defer p.mu.RUnlock()
	return &antennaPosition{
		platform:   p.platform.copy(),
		zero:       p.zero,
		head:       p.head,
		trueTarget: p.trueTarget,
		relative:   p.relative,
	}
}

func (p *antennaPosition) Position() common.Quaternion {
	p.mu.RLock()
	defer p.mu.RUnlock()
	return p.zero
}

// HeadPosition возвращает кватернион поворота антенны относительно носа платформы
func (p *antennaPosition) HeadPosition() common.Quaternion {
	p.mu.RLock()
	defer p.mu.RUnlock()
	return p.head.Multiplies(p.zero)
}

func (p *antennaPosition) TruePosition() common.Quaternion {
	p.mu.RLock()
	defer p.mu.RUnlock()
	return p.platform.TruePosition().Multiplies(p.head, p.zero)
}

// TargetPosition возвращает кватернион целевой точки антенны
func (p *antennaPosition) TargetPosition() common.Quaternion {
	p.mu.RLock()
	defer p.mu.RUnlock()
	return p.head.Conjugate().Multiplies(p.platform.TruePosition().Conjugate(), p.trueTarget)
}

func (p *antennaPosition) HeadTargetPosition() common.Quaternion {
	p.mu.RLock()
```

```
	defer p.mu.RUnlock()
	return p.platform.TruePosition().Conjugate().Multiplies(p.trueTarget)
}

func (p *antennaPosition) TrueTargetPosition() common.Quaternion {
	p.mu.RLock()
	defer p.mu.RUnlock()
	return p.trueTarget
}

func (p *antennaPosition) TrueMisalignment() float64 {
	return math.Abs(p.TrueTargetPosition().Dot(p.TruePosition()))
}

func (p *antennaPosition) HeadMisalignment() float64 {
	return math.Abs(p.HeadTargetPosition().Dot(p.HeadPosition()))
}

func (p *antennaPosition) Misalignment() float64 {
	return math.Abs(p.TargetPosition().Dot(p.Position()))
}

func (p *antennaPosition) Platform() Platform {
	return p.platform
}

func (p *antennaPosition) setPosition(q common.Quaternion) common.Quaternion {
	p.mu.Lock()
	defer p.mu.Unlock()

	p.trueTarget = p.platform.TruePosition().Multiplies(p.head, q)

	if p.relative {
		return p.zero.Conjugate().Multiplies(q)
	} else {
		return q
	}
}

func (p *antennaPosition) setHeadPosition(q common.Quaternion) common.Quaternion {
	p.mu.Lock()
	defer p.mu.Unlock()

	p.trueTarget = p.platform.TruePosition().Multiplies(q)

	if p.relative {
		return p.zero.Conjugate().Multiplies(p.head.Conjugate(), q)
```

```
	} else {
		return p.head.Conjugate().Multiplies(q)
	}
}

func (p *antennaPosition) setTruePosition(q common.Quaternion) common.Quaternion {
	p.mu.Lock()
	defer p.mu.Unlock()

	p.trueTarget = q

	if p.relative {
		return p.zero.Conjugate().Multiplies(p.head.Conjugate(),
p.platform.TruePosition().Conjugate(), p.trueTarget)
	} else {
		return p.head.Conjugate().Multiplies(p.platform.TruePosition().Conjugate(),
p.trueTarget)
	}
}

func (p *antennaPosition) updateCurrentPosition(q common.Quaternion) {
	p.mu.Lock()
	defer p.mu.Unlock()
	p.zero = q
}

func (p *antennaPosition) gyrostabilizer(pos inertial_navigation_system.Position) common.Quaternion
{
	p.mu.RLock()
	defer p.mu.RUnlock()
	_ = p.platform.updateCurrentTruePosition(pos)

	if p.relative {
		return p.zero.Conjugate().Multiplies(p.head.Conjugate(),
p.platform.TruePosition().Conjugate(), p.trueTarget)
	} else {
		return p.head.Conjugate().Multiplies(p.platform.TruePosition().Conjugate(),
p.trueTarget)
	}
}

func (p *antennaPosition) gyrostabilizerNewTrueTarget(truePosition common.Quaternion, pos
inertial_navigation_system.Position) common.Quaternion {
	p.mu.Lock()
	defer p.mu.Unlock()
	_ = p.platform.updateCurrentTruePosition(pos)
```

```
	p.trueTarget = truePosition

	if p.relative {
		return p.zero.Conjugate().Multiplies(p.head.Conjugate(),
p.platform.TruePosition().Conjugate(), p.trueTarget)
	} else {
		return p.head.Conjugate().Multiplies(p.platform.TruePosition().Conjugate(),
p.trueTarget)
	}
}

func (p *antennaPosition) updateTrueTarget(pos inertial_navigation_system.Position) {
	p.mu.Lock()
	defer p.mu.Unlock()
	old := p.platform.updateCurrentTruePosition(pos)
	p.trueTarget = p.Platform().TruePosition().Multiplies(old.Angles.Conjugate(), p.trueTarget)
}
```