UART üzenetek csomagolása

Kovács Viktor, kovacs.viktor@aut.bme.hu

2020.05.04.

Miért kell csomagolni?

Az UART nem definiál konkrét fizikai réteget, csak a logikai jelszinteket használja, elsősorban az átvitel időzítésére és egyszerű keretezésére koncentrál. A keretek néhány adatbitet (5-9) tartalmaznak, opcionálisan paritással is elláthatóak. Megbízható kapcsolat alapú átvitelt, vagy egyszerű keretek közötti hibadetektálást nem definiál, ez a felhasználóra van bízva. Az eszközök csatlakoztatása vagy szétcsatlakoztatása akár futás közben is megtehető, ezzel az adó és a vevő közötti szinkron sérülhet. Ugyanez igaz, ha eltérő időpontban kapcsoljuk be az eszközöket, valamelyiket újraindítjuk stb.

Az átvitel aszinkron, órajelet nem továbbítunk, a kommunikáció feltétele, hogy mind a fogadó, mind a vevő egymáshoz képest pontos és stabil órajellel rendelkezzen. Ennek az eléréséhez kvarc oszcillátorok javasoltak, de sok esetben a gyártók RC oszcillátorokat is megfelelő pontosságúra kalibrálnak, de ezt a hőmérséklet erősen befolyásolhatja.

Bár a fizikai réteget is definiáló RS232 vagy RS485 szabványok stabilabb átvitelt tesznek lehetővé, az időzítésből, csatlakozásból adódóan ezek sem tekinthetőek önmagukban teljesen megbízhatónak. Vessük össze ezt pl. a CAN üzenettel, ami több bájt átvitelére képes, tartalmaz hibadetektálást is.

Összefoglalva bájthiba, bájtok közötti szinkronizációs hiba előfordulhat, nem garantálható, hogy a bájtok a fogadó oldalon a vártnak megfelelően érkeznek be. Megbízhatóvá csomagolással lehet tenni, az adatfolyamot csomagokba tesszük, minden csomag vétele során meg kell tudunk győződni, hogy nem történt hiba a fogadása során.

0. Megoldás (egyszerű szöveges átvitel)

Gyakran használt megoldás, bár nem tekinthető megbízhatónak. Elsősorban csak az üzenetek határolása megoldott, pl. sorvége jelekkel. A határolást garantálja az újsor karakter, ez az adatrészben nem fordulhat elő, mert nem „olvasható” karakter. Hibaellenőrzést általában nem alkalmaznak (vannak kivételek, pl. gondoljunk az Intel .hex formátumra), ezért a vevő nem tud megbizonyosodni róla, hogy helyes sort fogadott. Platformtól függően a sorok végét egy vagy két karakter jelöli: Windows: ”\r\n” (0x0D, 0x0A), Linux: “\n” (0x0A). “\n”: új sor, “\r” kocsi vissza.

Figyeljünk erre, pl. hagyjuk figyelmen kívül a “\r” karaktert. Az üzenetküldés gyakoriságától függően érdemes lehet az üzenet elején is küldeni egy üres sort, sorhatárolót, így a vevő nem marad le az első üzenetről (ne értelmezze a töredék üzenetet!).

A küldés ebben az esetben egyszerű, opcionálisan küldhet az adó egy újsor szimbólumot, a szöveges üzenetet, majd egy lezáró újsor szimbólumot.

A vevő egy fogadó pufferbe fogadja az adatokat, ha bármikor újsor karakter érkezik, visszaállítja a számlálót az üzenet elejére. Figyeljünk, hogy fogadás közben ne írjunk a fogadó memória lefoglalt méretén túl. Ha ezt elértük, a fogadott üzenet biztosan érvénytelen lesz, várjunk a következő sorvége karakterre.

A megoldás **előnye** az egyszerűség, soros terminálban is könnyen olvasható vagy küldhető ilyen üzenet. Természetesen akár bináris adat is elküldhető pl. ASCII HEX formátumban.

A **hátrány**, hogy gyakran nincs hibaellenőrzés, bináris átvitelnél pazarló.

ASCII Code Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Csomagolás

A fenti példa egyszerű, de nem garantálja az átviteli hiba detektálását, valamint az átküldött adatokra nézve is megkötéssel jár, a küldőnek eleve garantálnia kell, hogy elsősorban szöveges adatokat küld, az adatcsomagban nem fordul elő a lezáró (újsor) karakter.

Az UART keret egy-egy bájt átviteléért és fogadásáért felel, ezért tartalmaz start bitet, adatmezőt, hibaellenőrzést és stop bitet. Hasonló felépítést célzunk meg a saját csomagolásunkkal is, de már az UART által továbbított bájtokat csomagoljuk egységbe.

A csomagok elejét (vagy az előző csomag végét) fel kell tudni ismerni. Leggyakrabban egy könnyen detektálható fejléccel, szinkronizáló mezővel indulnak a csomagok. A fejlécet akkor is fel kell tudni ismerni, ha hibás fogadás történt, „vissza lehessen találni”, újra szinkronizálódni az adóhoz. Kevéssé végiggondolt protokollokkal és adattartalommal előfordulhat, hogy sosem tud a vevő az adóhoz szinkronizálódni.

A csomagban átvitt adatmennyiség lehet fix vagy változó. Fix méretű csomagok előnyösek hasonló struktúrájú üzenetek esetén, ilyenkor nem kell fenntartani egy mezőt erre a célra. Sok esetben akár az is bevállalható, ha a leghosszabb előforduló üzenet hosszát használjuk és legrosszabb esetben nem használjuk ki az összes rendelkezésre álló adatot, memóriaszeméttel, de jobb esetben 0-val vagy más fix bájtokkal töltjük fel. Változó hosszúságú üzenet esetén a csomag elején definiálni kell egy mezőt, ami a vevő számára tudatja, hogy hány bájtot kell fogadni. Ez a mező lehet 8 vagy 16 bites is, de figyeljünk rá, hogy hiba esetén akár lényegesen nagy érték is olvasható itt, ezt kezeljük! Pl limitálhatjuk az üzenet hosszát 1000 bájtra, ha ennél nagyobb értéket olvasunk, már tudhatjuk, hogy hiba van és nem érdemes tovább fogadni, várjuk a következő csomag fejlécét.

A csomag végén érdemes mindig elhelyezni ellenőrzőkódot, ebből nagy valószínűséggel tudhatjuk, hogy hibátlan volt az üzenet fogadása. Az ellenőrzőkódot mindig a nyers elküldendő üzenetre számoljuk, mindenféle transzformáció előtt. Hibákat ellenőrizhetünk akár fogadás közben is, ha bizonyos mezők nem megengedett értéket vesznek fel: a fenti példában a hossz mező vizsgálata volt, de akár parancs bájtok is ellenőrizhetők.

Az ellenőrző kód lehet valamilyen paritás, ez nyújtja a legkisebb védelmet, ellenőrzőösszeg, vagy CRC. A CRC nagyon jó burst jellegű hibák detektálására, ami a kommunikáció során a leggyakoribb. Lehetőleg használjunk mindig CRC-t az előbbi megoldások helyett!

Használhatunk lezáró szimbólumot is, de nem nyújt sokkal nagyobb biztonságot, mint egy ellenőrzőösszeg.

A lentieken túl alkalmazhatunk kétirányú kommunikációt, nyugtázást az egyes csomagokhoz, illetve újraküldést, ha adott időn belül nem érkezett nyugta, de ez túlmutat ennek a dokumentációnak a célján. Ilyenkor célszerű csomag azonosító mezővel bővíteni az üzenetünket és minél egyszerűbben (pl. csomag azonosító legfelső bitje) kezelni a rövid nyugta üzenetet.

Fejléc, szinkronizációs karakter

A szinkronizációs karakter egy kitüntetett bájt vagy szimbólum. Bármilyen esetben, ha ezt detektálja a fogadó, valamilyen speciális funkciót vált ki, pl. üríti a fogadó puffert vagy befolyásolja a következő bájt értelmezését. Probléma, hogy mivel bármilyen adattömböt át kell tudnunk vinni a csomagban, nem zárható ki, hogy ez a bájt is előfordul az adattömbben. Mit lehet tenni?

Az egyik megközelítés: Ne forduljon elő az adatban! Ne használunk minden szimbólumot, vagy valamilyen transzformáció alapján érjük el, hogy az adatot kevesebb szimbólummal kódoljuk, ilyenkor az adat hossza akár változhat is. Ezekre a későbbiekben mutatunk példákat.

Különböztessük meg az adatoktól! A szinkronizációs karakter, mint vezérlőkarakter funkcionál, az utána fogadott bájttal együtt kap értelmet, ld. C string konstansok megadása. ”\n”, “\\”, “\r\n”…

A szinkronizációs karakterre mondhatjuk, hogy megsérti az adat kódolást, ld. „out of band” signaling.

1. Megoldás (4-bit nibble)

A szinkronizációs karakter ne fordulhasson elő az adatokban. Limitálhatjuk az átvitt hasznos bitek számát és fenntarthatjuk a szinkronizációs bájtot. A legegyszerűbb megoldás, hogy minden UART keretben csak 4 hasznos bitet viszünk át, pl. 0x00..0x0f-ig érvényesek a bájtok. Hasonló megoldás, ha 4-4 bit ASCII kódját visszük át, így akár szövegesen is olvasható, ha ránézünk az adatfolyamra. Azonban itt a karakterkódolás miatt külön kell vizsgálnunk a számok és betűk tartományát (0..9: 0x30..0x39, A..F: 0x41..0x46).

Lehetne akár 7 hasznos bitet is átvinni (érvényes adat: 0x00..0x7F), de ebben az esetben jóval bonyolultabb a transzformáció (régen nagyon sokat használták a modemes világban, az ASCII tábla is 7 bites). Számos ilyen kódolás létezik (pl. Base64, UUEncoding stb).

Illetve bizonyos esetekben, ha mindkét fél támogatja, lehet az UART keretet 9 bites módban is használni, ekkor az extra bit szolgálhat a szinkronizáció vagy egyéb vezérlés jelzésére, sajnos ezt nem minden hardver, szoftver támogatja.

Bár a 4 bites megoldás pazarló, ha nincs szükség nagy adatátviteli sebességre, vagy sok adat átvitelére, vállalható alternatíva lehet. Opció lehet akár az adatátviteli sebesség növelése is.

Válasszunk egy tetszőleges bájtot (nem az érvényes adattartományból), pl. 0xAA.

Az adó egyszerűen elküldi a kiválasztott szinkronizációs karaktert, majd sorban az egyes adatbájtok alsó és felső 4-4 bitjeit. Ha szükséges, akár az adathossz is elküldhető a szinkronizáció után. Végül az ellenőrzőösszeg is hasonló H-L transzformációval küldhető. Írhatunk saját UART transmit complete megszakítást, ami előállítja a kiküldött adatokat, vagy előre egy második pufferbe előkészíthetjük a csomagot küldéshez.

A vétel nagyon egyszerű, bájtosával fogadunk. Szükség van egy fogadó pufferre, egy változóra, ami megmondja épp alsó vagy felső 4 bitet fogadunk, valamint egy számlálóra, hogy hanyadik bájtot fogadjuk. Bármikor a szinkronizációs karaktert fogadjuk, a fogadó számlálót visszaállítjuk 0-ra, a HighLow változót is alaphelyzetbe állítjuk. Minden érvényes fogadott adatbájt esetén elmentjük a fogadó puffer számláló által mutatott helyére a HighLow változónak meglelően. A számláló akár a 4 biteket is számolhatja, eggyel leshiftelve a bájt címére is mutathat, ez részletkérdés. A HighLow változót mindig negáljuk. A fogadó puffert maximum a megadott értékig írjuk, utána eldobjuk a fogadott adatokat.

Az ellenőrző kód fogadása utána ellenőrizzük azonos algoritmussal a fogadott adatokat, egy jelző flag-et beállítunk a főciklus számára, hogy sikeres volt egy csomag fogadása. Amíg ez a változó aktív, eldobjuk a fogadott bájtokat. Az üzenet feldolgozása utána a főciklus törli a flag-et. Használhatunk akár több fogadó puffert is, így ha sokáig tart az üzenet feldolgozása, akkor is lehetséges a fogadás. Vagy a főciklus gyorsan lemásolhatja a puffert és jelezhet, hogy kész.

Teljesen mindegy, hogy milyen sorrendben küldjük az alsó vagy felső 4 bitet, de fontos, hogy az adó és a vevő ugyanazt a sorrendet használja.

A megoldás **előnye** az egyszerűség, főleg fogadás során, bináris adatokra is működik, **hátránya**, hogy 4 bites átvitel esetén pazarló, ha nem IT rutinban állítjuk elő a kimeneti bájtokat (alsó-felső), akkor egy újabb pufferbe kell transzformálni a csomagot küldés előtt.

Üzenet: 0xAC 0xDC 0xFF (crc8: 0x07)

Elküldött, csomagolt üzenet, Low-High sorrend

0xAA	0x0C	0x0A	0x0C	0x0D	0x0F	0x0F	0x07	0x00
SYNC	L(0xAC)	H(0xAC)	L(0xDC)	H(0xDC)	L(0xFF)	H(0xFF)	L(CRC)	H(CRC)

2. Megoldás (szinkronizációs bájt csere)

Válasszunk egy szinkronizációs bájtot, pl. 0xAA. Ha az üzenet maximális hosszát 253 bájtra maximalizáljuk (ellenőrzőöszeggel együtt), akkor garantáltan létezik legalább egy bájt, ami nem fordul elő az üzenetben, ez a csere bájt (pl. 0x42, de ez tartalom függő). Az üzenetben cseréljük le az összes

előforduló 0xAA bájtot erre a csere bájtra (ellenőrzőösszeggel együtt). Az üzenet lehet fix, vagy változó hosszúságú is.

Az adó küldje el a szinkronizációs bájtot, a csere bájtot, majd az üzenetet: 0xAA 0x42 […]

A vevő várja a szinkronizációs karaktert. Az ez után fogadott karaktert, a cserét mentse el. Ezután minden alkalommal az üzenet során ha csere bájt jön, a fogadó pufferbe írjon 0xAA-t.

A megoldás **előnye**, hogy egyszerű, nem változik az üzenet hossza, nem kell bájtokat beszúrni. **Hátrány**, hogy limitált az üzenet hossza, idő megtalálni a nem használt bájtot.

Üzenet: 0xAC 0xDC 0xAA 0xFF 0x00 0x40 (crc16: 0x4326)

0xAA	0x01	0xAC	0xDC	0x01	0xFF	0x00	0x40	0x26	0x43
SYNC	CSERE			0xAA!				L(CRC)	H(CRC)

3. Megoldás (vezérlőkarakter, byte stuffing)

Válasszunk egy tetszőleges vezérlőkaraktert, pl ’@’. Ha vezérlőkaraktert fogadtunk, máshogy értelmezzük a következő karakter(ek)t. Ld. C string literal ’\’ karaktere. Ez az eljárás módosíthatja a küldendő adat hosszát, előfordulhat, hogy extra bájtokat kell beszúrni az üzenetbe, hogy az eredeti tartalom visszaállítható legyen.

Az üzenet fejléce lehet egy speciális vezérlőszimbólum, pl. 0x3A (’:’). Ha az üzenetben előfordul a ’@’, akár többször is egymás után karakter, szúrjunk be még egyet: ”@@”. (Ez a CRC-re is vonatkozik! A CRC-t transzformáció előtt számítjuk és transzformáció után küldjük.)

Az adó egy másik pufferbe áttranszformálja az üzenetet küldés előtt. Akár egy külön függvénnyel előre meghatározhatjuk, mekkora lesz a kimenő üzenet mérete – ha ellenőrizni akarjuk, hogy belefér-e a cél pufferba, vagy ha dinamikusan akarunk memóriát foglalni, vagy hogy egyáltalán lesz-e szükség beszúrásra.

Fogadás esetén ha pl. 0x40-t fogadtunk (’@’), több ’@’ karakter esetén eggyel kevesebbet írunk a fogadó pufferbe. A vevő ilyenkor eltárolja, hogy vezérlő jött, ennek függvényében máshogy értelmezi a következő fogadott karaktereket. Ha nem vezérlő jön, törli a jelzést. A megoldás könnyen bővíthető, további speciális vezérlőkarakterek is használhatóak, definiálhatóak.

Figyeljünk, hogy a vevőben ne okozzon gondot az se, ha a CRC-ben 0x40 ’@’ bájt lenne! Esetleg küldhetünk tetszőleges extra nem vezérlőkaraktert az üzenet után, hogy biztosan ne okozzon problémát a vevőben!

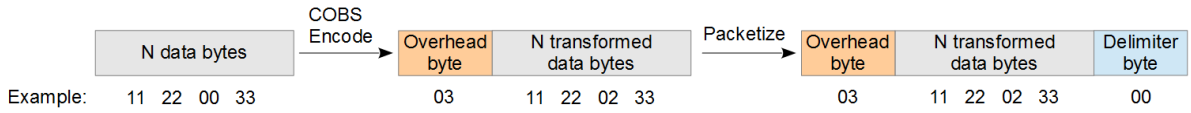
Hátrány, hogy változik a kiküldendő üzenet hossza, küldés előtt egy másik pufferbe kell átírni az adatokat, kicsit bonyolultabb a küldő és a fogadó.

Üzenet: 0xAC 0xDC 0x40 0x3A 0xFF 0x00 0x42 (crc16: 0xBF26)

0x40	0x3A	0xAC	0xDC	0x40	0x40	0x3A	0xFF	0x00	0x42	0x26	0xBF
’@’	’:’			’@’ !!!	’@’	’:’				L(CRC)	H(CRC)

4. Megoldás (COBS)

Válasszunk egy üzenet elválasztó karaktert, pl. „0x00”. Ezt el fogjuk küldeni minden üzenet végén, ez beszúrást jelent. Hogy ne legyen félreértés, az adatban előforduló 0x00-kat le kell cserélni. Hogyan? A csomag első, extrán beszúrt bájtja mondja meg, hogy hány karakter múlva következik a következő, az adatban 0x00-t reprezentáló karakter. Az üzenetben minden 0x00-t lecserélünk az utána következő 0x00 távolságára. Így az adatfolyamban beszúrást nem kell elvégezni, csak az elején és a végén.



https://en.wikipedia.org/wiki/Consistent_Overhead_Byte_Stuffing

Az adó elküldi az első 0x00 pozícióját (lezáró 0x00-val együtt), lecseréli az adatmezőben a 0x00-kat a relatív távolságukra majd végül küld egy 0x00-t. (Akár egy 0x00-val kezdheti is az első üzenet előtt.)

A fogadó vár az első 0x00 bájtra, ezután feljegyzi a következő bájtot egy számlálóba és minden fogadott bájtnál számol vissza, írja a fogadó puffert. Ha a számláló eléri a 0-t, a fogadó pufferbe 0-t ír és felírja újra a fogadó számlálóba a fogadott bájtot. 0x00 fogadásakor vége az üzenetnek.

Előny, hogy az adatmezőbe nem kell beszúrni, fixen mindig 2 bájttal lesz hosszabb az üzenet, mint az adattartalom.

Hátrány, hogy limitált az üzenet hossza, két speciális karakter közötti távolság nem lehet 255-nél nagyobb.

A megoldás kis fantáziával tetszőlegesen továbbfejleszthető, bájt beszúrással kezelhető akár hosszabb 0x00-t nélkülöző adatmező is (pl. megkülönböztetett 255-ös távolság számláló, ami beszúrt extra 0-ás karakterre mutat, amit a vevő nem ír be a pufferbe). A 0-s helyett használhatunk más speciális karaktert is, lehetőleg olyat, ami ritkábban fordul elő üzenetekben. Bináris üzeneteknél a 0 gyakorinak mondható.

Üzenet: 0xAC 0xDC 0x00 0xFF 0x00 0x40 (crc16: 0x9B07)

Küldött	0x03	0xAC	0xDC	0x02	0xFF	0x04	0x40	0x07	0x9B	0x00
Jelentés		0xAC	0xDC	0x00	0xFF	0x00	0x40	L(CRC)	H(CRC)	Vége
Számláló	3…	2…	1…	2…	1…	4…	3…	2…	1…	0

A protokoll máshogy is megközelíthető, a lezáró 0x00-t tekinthetjük első bájtnak is, ez nem változtat a működésen.

Ellenőrzőkódok

Számos ellenőrzőkód létezik, mint paritás, ellenőrzőösszeg, CRC (cyclic redundancy check), egyéb hash függvények. A CRC nagyon jó adatátvitel esetén, ahol a hiba burst-ösen jelentkezik. Lehetne bonyolultabb, hosszabb hash függvényeket is használni, de nem jár gyakorlati előnnyel.

A CRC-nek számos eltérő változata van: eltér a kód hossza (pl. 4bit, 5 bit, 8 bit, 12, 16, 32…), a generátor poilnom, a kezdőérték, valamint a be és kimenete is negálható. Nagyon sok szabvány van, ezek a fenti paraméterekben térnek el. Ne találjunk ki saját generátor polinomot, használjunk egy olyat, amit más szabványban is alkalmaznak. Ez hatással lehet a hibadetektálásra. Üzenet hosszától függően ajánlott CRC8, 16, 32 használata.

C implementáció nagyon egyszerűen működésre bírható, a gyors implementációk LUT táblázatokat használnak. A legtöbb STM32 mikrovezérlő tartalmaz univerzális CRC számoló hardvert, ez is jól használható.

- https://en.wikipedia.org/wiki/Cyclic_redundancy_check
- http://www.sunshine2k.de/coding/javascript/crc/crc_js.html
- http://www.sunshine2k.de/articles/coding/crc/understanding_crc.html
- https://crccalc.com/
- https://github.com/lammertb/libcrc/tree/master/src
- C Crc32: https://opensource.apple.com/source/xnu/xnu-1456.1.26/bsd/libkern/crc32.c

Fogadás mikrokontrollerrel

Fogadjunk bájtosával, mert előre nem tudjuk, ténylegesen mennyi bájt fog érkezni. Használjunk megszakítást!

Sikeres csomag vétel esetén állítsunk be egy flag-et, amit a főprogram is elér és ciklikusan ellenőriz, hogy van-e fogadott adat. Biztosítsunk a főprogram számára elérhető függvényt, amivel megkapja az üzenet kezdőcímét és hosszát. Amíg a főprogram nem törli a flag-et, ne fogadjunk ugyanerre a címre. Akár a főprogram is biztosíthatja az UART fogadó számára a puffert. Akár több puffert is használhatunk fogadáshoz, amit sikeres fogadás után cserélhet a fogadó.

Összetettebb protokollok esetén használjunk állapotgépet! Példa állapotok: fejlécre vár, adathossz érkezik, adatmező… A fenti protokollok elég egyszerűek.

Határozottan szebb megoldás, ha a megszakításkezelő a lehető legrövidebb, pl. csak pufferbe menti az adatokat. Érdemes egy cirkuláris pufferbe (fifo-ba) írni az adatokat a megszakításkezelővel és a főciklusban kiolvasni és bájtosával a fenti protokollok dekóderével feldolgozni.

A HAL jó a perifériák konfigurálására, a fogadás indítására, de nagyon pazarló bájtos fogadáshoz. A bájt sikeres fogadása esetén ki is kapcsolja a megszakítást amit minden alkalommal újra be kell kapcsolni. Meg lehet csinálni, hogy lemásoljuk a HAL megszakításkezelőjét és kiirtjuk belőle a felesleges kódot, a natív megszakításkezelőben ebbe a saját változatba hívunk vissza. Ezt csak annak javasoljuk, aki tudja mit csinál és tisztában van a következményekkel.

Ezen kívül szép megoldás DMA-val, „hardveresen” fogadni cirkuláris pufferbe, időnként ellenőrizni az író és olvasó pointer eltérését, ehhez megszakításra sincs szükség. A TC és HT (Transmit Complete, Half-Transfer Complete) megszakítások jók túlcsordulás kezelésére, duplapufferelésre vagy a főprogram értesítésére, hogy ideje olvasni. A főprogramnak garantálni kell, hogy két ellenőrzés között nem tud betelni a fogadó puffer.

https://stm32f4-discovery.net/2017/07/stm32-tutorial-efficiently-receive-uart-data-using-dma/

Az adatmező gyakran egy üzenettípus vagy parancs mezővel kezdődik, ez meghatározhatja az üzenet hosszát is, ilyenkor érdemes egy táblázatból kinézni a típushoz tartozó hosszt. Így elkerülhető a külön üzenethossz mező.

Fogadás PC oldalon

SerialPort osztály nagyon kényelmes, nagyon egyszerű a küldés. Javasolt még megnézni a MemoryStream és BinaryWriter segéd osztályokat.

A fogadás esetén figyelni kell, hogy a fogadás esemény más szálról hív vissza, erről a szálról nem férhetünk hozzá közvetlenül a felhasználói felület objektumaihoz. Ilyenkor egy Windows üzenettel szólhatunk a GUI-t futtató szálnak, hogy hajtson végre egy műveletet. Ez az Invoke-kal tehető meg. Az Invoke-minta alkalmazásával bármilyen szálról hívhatunk GUI-t elérő műveleteket. Az invoke számos módon használható, a nyelvi elemek bővülésével egyre egyszerűbb szintaktikával.

```
public void ChangeText(string text)
{
    if (tbSzoveg.InvokeRequired)
    {
       tbSzoveg.Invoke(new Action<string>(ChangeText),new[] { text });
       //vagy
       //tbSzoveg.Invoke(new MethodInvoker(() => { ChangeText(text); }));
       //vagy…
    }
    else
    {
        tbSzoveg.Text=text;
    }
}
```

vagy tömörebben, Extension Method-dal és Action-nel:

```
public static void InvokeIfRequired(this Control c, Action<Control> action)
{
    if(c.InvokeRequired)
    {
        c.Invoke(new Action(() => action(c)));
    }
    else
    {
        action(c);
    }
}
```

https://docs.microsoft.com/en-us/dotnet/framework/winforms/controls/how-to-make-thread-safe-calls-to-windows-forms-controls

Adattartalom

Az adatmező tartalma lehet bináris vagy szöveges, a bináris tömörebb, de általában kevésbé univerzális.

Bináris esetben érdemes protokoll verziót mezőt is felvenni, hogy a vevő tudja, hogy kell értelmezni az adatokat, főleg visszafele kompatibilitás miatt, hogy régebbi fogadót ne zavarjon össze újabb adó. Bináris protokolloknál általában kötött az adatok sorrendje, a sorrend (cím) határozza meg az adatok értelmezését, jelentését, így kevésbé rugalmasan kezelhető. A hátránya egyben előnye is, mert a fogadott puffer egyszerűen egy struktúrává cast-elhető és az elemei azonnal elérhetőek. Változó hosszúságú listák esetén az elemek számát is meg kell adni a lista előtt. Ez nehezebben kezelhető struktúrává cast-eléskor. Ugyanígy gondot okozhat eltérő architektúrák esetén az eltérő bájtsorrend (pl. x86, ARM little endiant használ, de az is előfordulhat, hogy az adó bájtonként, manuálisan állítja elő a kimenetet). Ilyenkor kössük ki, hogy little endian vagy big endian szerint tárol a protokoll. A szóhossz illesztés is gondot okozhat, több architektúra esetben pl. float csak szóhossz határon lehet (néggyel osztható memóriacímen). Hasonló okokból a struktúrákban a mezők illesztett (aligned) címekre kerülnek a típusuktól és sorrendjüktől függően, így keletkezhetnek hézagok, ami kommunikáció során pazarlás. Struktúra definiálásakor megadható a *packed* attribútum, ilyenkor nem lesznek üres területek, de lassabb lehet a hozzáférés. Fontos, hogy mindkét oldalon használjuk a packed attribútumot.

```
struct __attribute__((packed)) {
	uint8_t msgType;
	uint32_t msgLen;
	//…
}
```

A protocol buffer (protobuf) egy tömör, univerzális szinte minden platformon elérhető csomagoló, sorosító kifejezetten kommunikációs célokhoz.

https://developers.google.com/protocol-buffers

A szöveges JSON mintájára léteznek bináris univerzális csomagolók, pl. CBOR vagy a BSON.

A szöveges formátumok univerzálisabbak, az egyik legelterjedtebb a JSON, kulcs-érték párok, összetett típusok (objektumok) és tömbök tárolására szolgál. Minden platformon elérhető, de nem nevezhető nagyon tömörnek. Kompaktabb változata a MessagePack. Korábban az XML volt nagyon elterjedt, de ma már a JSON-t használnak, ahol csak lehet.

https://arduinojson.org/

https://msgpack.org/